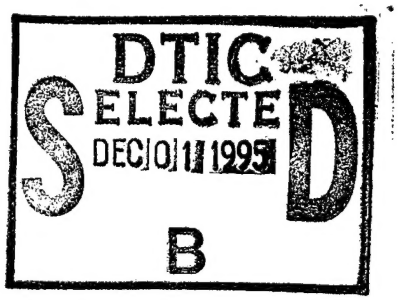
ARMY RESEARCH LABORATORY

Force Wars, A Model of Army Systems in Combat Including Fratricide

Fred L. Bunn

ARL-TR-883 October 1995



NOTICES

Destroy this report when it is no longer needed. DO NOT return it to the originator.

Additional copies of this report may be obtained from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The use of trade names or manufacturers' names in this report does not constitute indorsement of any commercial product.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1995	Final, October 1993 - June 1995

4. TITLE AND SUBTITLE
Force Wars, A Model of Army Systems in Combat Including Fratricide

5. FUNDING NUMBERS
1L162618AH80

6. AUTHOR(S)
Fred L. Bunn

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-WT-WE
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-883

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes a stochastic simulation of combat between Army weapon systems. It is oriented toward those who are interested in using the model to perform studies of the combat effectiveness of weapon systems and their subsystems and toward those who are interested in extending the model.

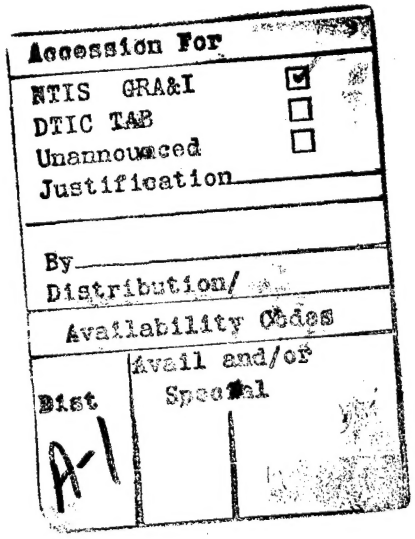
The model was designed to analyze combat effectiveness of Army systems as well as their subsystems. It is an extension of the Tank Wars model. Major extensions include fratricide, a mix of weapon systems on each side and multiple kinds of weapons/rounds on each platform. It can simulate combat of as many as 50 combatants of 10 kinds, but this is easily increased. The model is written in FORTRAN 90 using top-down structured programming. It is based on the old Tank Wars model which is running at more than a dozen installations.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Tanks (Combat Vehicles), Warfare, Simulation, Armored Vehicles, Materiel	80
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

INTENTIONALLY LEFT BLANK.

CONTENTS



1. INTRODUCTION

The Force Wars model simulates combat between a mix of systems on each side. Currently, it handles as many as 50 systems which may be of as many as ten types. These are easily increased by changing two parameters in the code. The types of systems include tanks and armored fighting vehicles (AFVs). It was written with the intention of adding artillery, Interim Tow Vehicles (ITVs), helicopters, fixed wing, and dismounted infantry at a later time. Each system may be armed with one to three types of weapons or ammunition. Any of them may be a gun or a fire and forget missile. The first weapon may be a missile system capable of launching simultaneous missiles and guiding them to impact.

Motion. The motion of the systems is rather simplistic now. Each is given an initial position in three dimensions, and the attackers are given a horizontal velocity. Every 5 seconds, the model updates the position of surviving attackers. Since ground vehicles move about 5 m/s during combat, they will travel perhaps 25 meters between updates. This resolution appears satisfactory for ground vehicles.

Contact. Every 5 seconds, the model checks to find when each sytem comes into contact with other systems. This can be accelerated greatly in the future.

The model finds when a target enters or exits maximum detection range of a system that is searching. When the target enters this range, search begins. When the target exits this range, the firer ceases to engage the target.

The range to which the searcher detects depends on whether the target is fully exposed and moving, fully exposed and stationary, in hull defilade and stationary, or in turret defilade and stationary.

Exposure and Line-of-Sight. Force Wars models exposure caused by three factors: pop-up/pop-down defensive tactics, bounding overwatch tactics of the attacker, and the mask/unmask by terrain as entities move. At the beginning of the battle, every system is set to fully exposed, then defenders are set to turret defilade.

After a target comes into range of a firer, Force Wars checks to see if line-of-sight exists between them. This occurs if both are partially or fully exposed and there is no terrain blocking the line of sight.

Search. If any undetected target is within detection range, the search submodel is scheduled once each second. This continues while such targets are undetected. When it finds that a firer will detect a target in the next second, it schedules the target to be detected at a random time in the 1-second interval.

Classification. Search then calls **clasif** to classify the system as a friend or foe. This is the key routine for fratricide because it models the small chance that friends will be mis-classified as foes. It also mis-classifies foes as friends with a small probability. If the firer identifies the target as a foe, the **select** routine is called.

Target and weapon/ammunition selection. When a firer makes its initial selection, it selects the target and the weapon/ammunition combination that yields the

greatest kill probability. It will continue with this selection until disengagement, as discussed later.

The only exception to this is for Kinetic Energy Missiles (KEMs) or similar missile systems that have multiple guidance channels. If the firer is guiding a missile toward a target and has other guidance channels that are idle, it is able to select another target and fire at it while simultaneously guiding toward the first target. In this case, the firer selects a new target but uses the same type of weapon/ammunition. It selects the unengaged target with the highest kill probability. This is a reasonable criterion but not the only one.

Firing. When firing occurs, the model reduces the number of rounds on board by one, schedules an impact event, and calls the **pinpoint** routine to see if any foes detected the firer by its muzzle flash or smoke.

If the round is a fire-and-forget round, the firer has several options after firing. If he is completely out of ammunition, he disengages the target and hides. Otherwise, if he meets the pop-down criteria, he pops down. This happens if his mission is to defend and he has fired the appropriate number of rounds during pop-up (often two rounds). Otherwise, if he meets the switch targets criteria, he disengages the current target and attempts to select a new one. This is also based on the number of rounds he has fired at the current target. If none of these conditions hold, the program schedules the firer to fire again at the same target after a delay for reloading.

If the round is a guided round, the firer has similar options after firing. Since at least one round is being guided, however some of the options cannot be exercised immediately after firing. It can fire another round at the same target (wasteful) or another round at a new target, but only if it has a free guidance channel.

The options of hiding and popping down are delayed until impact. The options of firing at a new target or firing again at the same target are delayed until impact if all guidance channels are in use.

Impact. Impact checks to see if the target is exposed. If so, firer i disengages target j and calls **damage**, which finds if the target died. If the firer had fired a guided missile, **fired_missile** is called to handle subsequent actions.

Damage. The **damage** routine finds whether the target is killed. If it is killed, **damage** cancels all events the target was scheduled to perform and marks it as dead. It also calls **diseng** to cause all firers engaging the dead target to switch to a new target.

Damage uses probability of kill given a shot (PKS) data. Assuming there are 10 kinds of weapons, the data consist of information for these combinations:

$$(10 \text{ firers})(10 \text{ targets})(3 \text{ weapons})(3 \text{ exposures}) = 900 \text{ combinations}$$

For each combination, the data consist of 7 PKS values, for 0 to 3 km in 500-meter increments. This is hard wired and must be improved to allow for short range weapons, longer range weapons, and perhaps artillery.

Disengagement. The **diseng** routine handles target disengagement. Under the appropriate conditions, the firer(s) select new targets. This is the most difficult portion of the code to get right. Disengagement can occur because of many reasons. The firer may disengage because it popped down, ran out of ammunition, or because of a policy to switch targets after a fixed number of shots at the current target. If the target popped down, died, went behind terrain, or went out of range, the firer will also disengage.

2. USING FORCE WARS

This section is a users' manual for the model. It describes the input and output as well as the error messages generated.

2.1 Installation

Contact Fred Bunn at (410) 278-6676 if you wish a copy of the code. We prefer to distribute it on IBM compatible floppy disks. The disk contains the code, documentation, and five test cases. Each test case consists of a set of input files and output files.

The program is written in Fortran 90 and should run on any computer having a compiler for that language.

2.2 Input

The program requires four input files as follows:

d.kinds Describes each kind of system
d.sense Describes the sensing of each kind of system
d.kill Describes the lethality of each system versus each system
d.move Describes the motion of each system

System descriptions. The **d.kinds** file describes kinds of systems. Here is a sample of what the file looks like:

```
><    item to print for demo or debug
1     1     0     1     1111111     1999.0      # of reps, etc
8     # of kinds of systems
M60   M1    T72   T80   BFV   Apache      Hind  Inf
3     3     3     3     2     1     1     2     # weapons
55    39    40    39    30    30    30    5000  Wpn 1 ammo
200   200   200   200   200   0     0     200   Wpn 2 ammo
5000  5000  5000  5000  0     0     0     0     Wpn 3 ammo
10.   10.   10.   10.   10.   10.   10.   10.   tbr
2     2     2     1     2     2     2     2     # guidance channels for wpn 1
1.    1.    1.    1.    1.    1.    1.    1.    vm - 1
1.    1.    1.    1.    1.    1.    1.    1.    vm - 2
1.    1.    1.    1.    1.    1.    1.    1.    vm - 3
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   beta - 1
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   beta - 2
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   beta -3
9.9   9.9   9.9   9.9   9.9   9.9   9.9   9.9   tfirst
0.3   0.3   0.3   0.3   0.3   0.3   0.3   0.3   pinpnt?
1     1     1     1     1     1     1     1     loader type?
2     2     999   999   999   999   999   999   Rounds shot until popdown
2     2     3     3     3     3     3     3     Rounds shot until switch
```

Line 1. This line controls debug printing. The program reads as many as six characters from this line. If it doesn't recognize the characters, it doesn't print any debug information. The values it recognizes and the action it takes are as follow:

'><' Use this to print every time a routine is entered and exited. A line will be

printed to file **output** in the current directory. It has the form '>xxx' when routine xxx is entered and '<xxx' when routine xxx is exited. It tells when each routine is entered and exited and can be used to check that the program is behaving properly or indicate where the program died.

The values: 'vanish', 'clock ', 'fire', 'meet', 'move', 'search', or 'sense ' are the names of routines. If one of them is used on Line 1, it causes values in those routines to be printed to standard output.

Line 2.

Value 1, the number of battles (replications) to be performed.
Value 2, the firer to monitor. A zero means monitor all.
Value 3, the target to monitor. A zero means monitor all.
Value 4, the replication to monitor. A zero means monitor none.
Value 5, the random number seed. Use 1111111 unless you have a better value.
Value 6, the maximum time for a battle to run. This protects against infinite loops.

Line 3. The value on this line is the number of kinds of systems to be described in later rows. In the succeeding lines, there is one value for each kind of system.

Line 4. The names of the systems. Each name is six characters or less.

Line 5. The number of weapons on each.

Line 6. The number of rounds of ammunition for Weapon 1 on each.

Line 7. The number of rounds of ammunition for Weapon 2 on each.

Line 8. The number of rounds of ammunition for Weapon 3 on each.

Line 9. The time between rounds for each.

Line 10. The number of guidance channels for Weapon 1 for each. If the value is zero, the weapon fires fire-and-forget rounds.

Lines 11-13. The muzzle velocity for weapons 1 through 3 on each.

Line 14-16. The Siacci drag coeffient for weapons 1 through 3 on each.

Line 17. The mean time to fire the first round for each (s).

Line 18. The loader type for each. (1 = manual loader, 2 = load assist, 3 = auto-loader).

Line 19. Rounds to fire at a target before the defender pops down. If the system is not a defender that uses pop-up and pop-down tactics, use a large value such as 999.

Line 20. Rounds to fire at a target before switching targets. U.S. tank tactics are to shoot until a kill, so this should be a large value for them. Missiles are expensive and often quite lethal. One is a good value to use for them.

Lines 21-50. Not currently read. For future use.

Combatant descriptions. The input file **d.move** contains data for individual combatants. The program ignores the first two lines. The third line contains information about Combatant 1, the fourth about Combatant 2, and so on. The file may contain 50 + 2 lines describing 50 individual combatants. The last line describing a combatant must be followed by a line with the characters 'END' in columns 1 through 3. Any subsequent lines are ignored and may be used for comments. The sample file below is for a battle among only three combatants.

```
DEPLOYMENT
SYS SIDE BUDDY X Y Z MISSION VMAX VX VY
M1     1 0 -0.1 -0.1 0. Defend 6.0 5.0 0.0
T80    2 3 3.1 -0.4 0. Attack -6.0 -5.0 0.0
T80    2 2 3.1 -0.2 0. Attack -6.0 -5.0 0.0
END
F/Run/d.move
```

Value 1. The name of the kind of system may be as many as six characters, e.g., 'M1A1', 'Apache'. Don't include the quote characters in the input. This allows the program to associate values in the **d.kinds** file for kinds of systems with individual combatants.

Value 2. The side the combatant is on. It should be either 1 or 2.

Value 3. The ID of the combatant's buddy. U.S. tanks, for example, use a pop-up/pop-down tactic in defense where a tank and his buddy alternate exposures. In attack, a pair of tanks often alternate between overwatch and bounding forward. If combatants one and two are buddies, this value will be 2 on Line 1 and 1 on Line 2.

Values 4-6. The starting coordinates of the combatant. List them in this order: distance east, north, and up from some arbitrary origin in kilometers.

Value 8. The maximum speed of the combatants (m/s). This is used for finding when combatants encounter each other.

Values 9, 10. This is the speed of the combatants Easting and Northing (m/s). These values are read by subroutine **detect_rg**. In the future, it may be expanded to handle

paths of the combatants that are more complicated than single straight lines. These more complex paths may take the form of complex pre-planned paths or dynamic paths that adjust to the situation the combatants encounter.

Sensing data. Sense reads these data from file **d.sense**. The first line of the file is for information and is skipped by the program. If there are 10 kinds of systems, then there must be 10 x 10 subsets of sensing data. Each subset looks something like this:

```
7 M60 M60
Rg  S-HD S-FE M-FE  S-HD  S-FE  M-FE Pfoe
0.0 0.99 1.00 1.00 0.014 0.019 0.018 0.00
0.5 0.70 0.90 0.95 0.008 0.015 0.015 0.01
1.0 0.40 0.82 0.90 0.003 0.009 0.012 0.02
1.5 0.25 0.65 0.80 0.002 0.007 0.009 0.04
2.0 0.15 0.45 0.60 0.002 0.005 0.006 0.05
2.5 0.05 0.15 0.20 0.001 0.002 0.003 0.06
3.0 0.00 0.00 0.00 0.000 0.000 0.000 0.07
```

The integer on the first line tells how many rows of data follow. **Sense** ignores the system names on that line and the next line that contains column headings. The data in the subsequent rows are for increasing distances between the firer (sensor) and the target.

Value 1. Distance (km).

Value 2. The probability of ever detecting a stationary, hull defilade target.

Value 3. The probability of ever detecting a stationary, fully exposed target.

Value 4. The probability of ever detecting a moving, fully exposed target.

Value 5. The probability of detecting a stationary, hull defilade target in the next second, given that it can be detected.

Value 6. The probability of detecting a stationary, fully exposed target in the next second, given that it can be detected.

Value 7. The probability of detecting a moving, fully exposed target in the next second, given that it can be detected.

Value 8. The probability of identifying the target as a foe.

Kill probabilities. Input reads the kill probabilities from file **d.kill**. The first line of the file contains an integer telling how many ranges (columns) of data are in the file. It reads only the integer. The rest of the line is used for a comment. If there are eight combatants, there will be 8 x 8 = 64 subsets of data. There are three lines of data in the data subset, one for each of the three weapons that the combatant may have available. There is a header line, so there are 1 + 3 x 64 = 193 lines of data in the file. When the **damage** routine is extended to treat more conditions, the data will have to be

increased to give kill probabilities for those conditions. Currently, the ranges are hardwired into the code and are for 0 to 3 km in half- kilometer increments, and there are seven columns of data for the seven ranges.

The example below show a dummy **d.kill** file with many subsets removed.

```
7  Ranges
   0.800   0.734   0.566   0.367   0.200   0.092   0.036  M60     M60     1
   0.700   0.626   0.449   0.258   0.118   0.044   0.013  M60     M60     2
   0.000   0.000   0.000   0.000   0.000   0.000   0.000  M60     M60     3
   0.800   0.734   0.566   0.367   0.200   0.092   0.036  M60     M1A1    1
   0.700   0.626   0.449   0.258   0.118   0.044   0.013  M60     M1A1    2
   0.000   0.000   0.000   0.000   0.000   0.000   0.000  M60     M1A1    3
   (18 lines for kind 1 vs kinds 3-7)
   0.000   0.000   0.000   0.000   0.000   0.000   0.000  M60     Inf     1
   0.000   0.000   0.000   0.000   0.000   0.000   0.000  M60     Inf     2
   0.700   0.350   0.044   0.001   0.000   0.000   0.000  M60     Inf     3
   (24 lines for kind 2 vs kinds 1-8)
   (24 lines for kind 3 vs kinds 1-8)
   (24 lines for kind 2 vs kinds 1-8)
   (24 lines for kind 5 vs kinds 1-8)
   (24 lines for kind 6 vs kinds 1-8)
   (24 lines for kind 7 vs kinds 1-8)
   (24 lines for kind 8 vs kinds 1-8)
```

2.3 Output

2.3.1 Input Echo The program opens the file **output** and prints an echo of the input from file **d.kill**. Then it opens file **d.move** and echos the names and positions of the actual combatants. Here is what an input echo looks like:

```
INPUT ECHO:
Replications              1
# kinds of comba          8
Name of kind          M60       M1       T72       T80       BFV    Apache     Hind      Inf
# of weapons             3        3         3         3         2         1        1        2
Weapon 1 ammo           55       39        40        39        30        30       30     5000
Weapon 2 ammo          200      200       200       200       200         0        0      200
Weapon 3 ammo         5000     5000      5000      5000         0         0        0        0
Time between rds     10.00    10.00     10.00     10.00     10.00     10.00    10.00    10.00
Guidance channel        2        2         2         1         2         2        2        2
Muz vel 1 (km/s)     1.00     1.00      1.00      1.00      1.00      1.00     1.00     1.00
Muz vel 2 (km/s)     1.00     1.00      1.00      1.00      1.00      1.00     1.00     1.00
Muz vel 3 (km/s)     1.00     1.00      1.00      1.00      1.00      1.00     1.00     1.00
Wpn 1 beta (hz)      0.00     0.00      0.00      0.00      0.00      0.00     0.00     0.00
Wpn 2 beta (hz)      0.00     0.00      0.00      0.00      0.00      0.00     0.00     0.00
Wpn 3 beta (hz)      0.00     0.00      0.00      0.00      0.00      0.00     0.00     0.00
Tfirst               9.90     9.90      9.90      9.90      9.90      9.90     9.90     9.90
Pinpnt               0.30     0.30      0.30      0.30      0.30      0.30     0.30     0.30
Loader type             1        1         1         1         1         1        1        1
Rds to pop-down         2        2       999       999       999       999      999      999
Rds to disengage        2        2         3         3         3         3        3        3
# of actual combatants      3
  M1       T80      T80
       1        2        2
```

```
  -0.10    3.10    3.10
  -0.10   -0.40   -0.20
   0.00    0.00    0.00
```

2.3.2 Summary Output Here is what the summary output looks like:

```
 SUMMARY STATISTICS FOR ALL BATTLES

          Firer    1
         Target    1    2    3
In range           0    8    3
Exit range         0    7    2
In LOS             0    8    3
Detected           0    1    1
Pinpoint           0    0    0
ID as friend       0    0    0
ID as foe          0    1    1
Engaged            0    1    1
Fired on           0    1    1
Killed             0    1    1

          Firer    2
         Target    1    2    3
In range           0    0   11
Exit range         0    0   10
In LOS             0    0    9
Detected           0    0    1
Pinpoint           0    0    0
ID as friend       0    0    1
ID as foe          0    0    0
Engaged            0    0    0
Fired on           0    0    0
Killed             0    0    0

          Firer    3
         Target    1    2    3
In range           0    1    0
Exit range         0    0    0
In LOS             0    3    0
Detected           0    2    0
Pinpoint           0    0    0
ID as friend       0    2    0
ID as foe          0    0    0
Engaged            0    0    0
Fired on           0    0    0
Killed             0    0    0
 Who scored first kill:
   1    0    1    1
   2    0    0    0
   3    0    0    0
 Who scored a kill:
   1    0    1    1
   2    0    0    0
   3    0    0    0
#dead on side 1, side 2, exchange ratio=     0     2     0.00

  DEAD   OUT  INRG   LOS  SEEN  FREN   FOE   TGT  <- Old values
     0     0     0     1     0     0     0     2 DEAD
     0     0     6    12     0     1     0     0 ISOUT
     0     0     0     4     0     0     0     0 INRG
     0     0     0     0     0     0     0     0 INLOS
     0     0     0     0     0     0     0     0 ISSEEN
     0     0     0     0     0     0     0     0 ISFREN
     0     0     0     0     0     0     0     0 ISFOE
```

```
   0    0    0    0    0    0    0    0 ISTGT
SUSPICIOUS:
   0    0    6   12    0    1    0    0 Now ISOUT
   1   12    4    0    0    0    0    0 Was INLOS
```

Final status of connections. The first piece of summary information is a matrix showing the connection between combatants. There is one row for each firer and one column for each target. Normally, the entries are zeros except on the diagonal. Along the diagonal, a 1 means the system is not searching and a 3 means it is searching for targets. Off the diagonal, possible values are:

0	Ignore this combination, firer or target is dead.
1	Target is alive, check to find when it comes in range of firer.
2	Target is in detection range of firer, check to find when LOS exists.
3	Target is in LOS of firer, find when it is detected.
4	Target is detected by firer, classify it.
5	Firer considers target a friend.
6	Firer considers target a foe, it is available for selection.
7	Firer has selected target for engagement.

Summary statistics for each firer. The next lines record the number of times certain events occurred between a firer and each of its targets. In the example above, there is information about Firer 1 and Targets 1 through 3. Since Firer 1 never treats itself as a target, the column for Target 1 contains all zeros. In the next column, we see that Target 2 was in range 8 times and out of range 7 times. This occurred because 2 was moving and line of sight was broken 7 times. Firer 1 detected Target 2 one time, classified it as a foe once, engaged it once, fired on it once, and killed it once.

Who killed whom. The next matrix shows who made the first killing shot on each target. Again, there is a row for each firer and a column for each target. In the example above, Firer 1 scored the first kill on Targets 2 and 3 as indicated by the 1's on the first row. The matrix after that shows who scored any kill even if the target was already dead.

Count of kills and exchange ratio. The next line simply tells how many were killed on sides one and two and gives the exchange ratio. The exchange ratio is set to zero if no combatants on side one were killed. This avoids a divide by zero. Otherwise, the exchange ratio is the number killed on side two divided by the number killed on side one.

Count of connection changes. The next matrix has a row and column for each of the eight possible connections between firer and target. The entries in the matrix give a count of the number of times that a connection was changed from a specific value to another specific value.

In the example above, Line 1 shows that twice systems were being engaged and became dead.

2.3.3 Event History When the fourth value on the second line of input in the **d.kinds** is set to a non-zero value, the program produces an event history for that battle. For example, if it is set to 100, the 100th battle will be shown. This assumes that at

least 100 battles are played.

The event history looks like this:

```
Replication 1
In view and out of view segment lengths (m):
 In Out  In Out  In Out  In Out  In Out  In Out  In Out  In Out  In Out  In Out
434 320 489 900 333 541  17 900 862 861  55 605 215 430 119 304  43 900 358 900
207 176 311 441 267 781  81 499 105 720 900 390  93 306 382 900   8 900  17 900
119 500 316 152 435 253  13 585 300 628 289 174  11 900 574 315  85 702 459 345
115 270 420 659 854 900 147 900 239 869 547 126 349 492  88 858 284 372 225 592
 52 494  97 326 565 161  23 690  81 900 819 900 237 375  16 592 209 512 454 680


 3  0   0.00          bound  event          :
 3  0                 bound  bound      0   0 1785.00       search events
 2  0   0.00          bound  event      0   0 1786.00       search events
 2  0                 owatch bound      0   0 1787.00       search events
 3  0   0.00          vanish event      0   0 1788.00       search events
 3  0                 doesnt vanish     1   2               detect search
 0  0   0.00          meet   event      1   2               enemy! clasif
 2  3                 convrg meet       1   2               select select
 2  3                 have   los        0   0 1790.00       move   events
 3  2                 convrg meet       0   0 1790.00       meet   events
 3  2                 have   los        1   2 1790.00       popup  events
 3  0   0.00          vanish event      1   2               does   popup
 3  0                 doesnt vanish     0   0 1795.00       move   events
 0  0   0.00          search event      0   0 1795.00       meet   events
 0  0   1.00          search event      1   2 1796.95       fire   events
 0  0   2.00          search event      1   2               fire @ fire
 0  0   3.00          search event      1   2 1798.26       impact events
 0  0   4.00          search event      1   2               (have) impact
 0  0   5.00          move   event      1   2               kills  damage
 0  0   5.00          meet   event      1   2               drops  diseng
 0  0   5.00          search event
                      :                 1   0 1798.26       popdn  events
 1  3                 detect search     1   0               does   popdn
 1  3                 enemy! clasif     0   0 1798.26       endsim events
 1  3                 select select
 0  0  961.67         search event
 1  3  962.67         popup  event
 1  3                 does   popup
 0  0  962.67         search event
 0  0  963.67         search event
 0  0  964.67         search event
 0  0  965.00         move   event
 0  0  965.00         meet   event
 0  0  965.67         search event
 0  0  966.67         search event
 0  0  967.67         search event
 0  0  968.67         search event
 0  0  969.67         search event
 0  0  970.00         move   event
 0  0  970.00         meet   event
 0  0  970.67         search event
 3  2                 detect search
 0  0  971.67         search event
 0  0  972.67         search event
 1  3  973.22         fire   event
 1  3                 fire @ fire
 0  0  973.67         search event
 1  3  973.97         impact event
```

```
 1  3                (have) impact
 1  3                kills  damage
 1  3                drops  diseng
# survivors = 1 1 # targets = 0 0
 1  0  973.97        popdn  event
 1  0                does   popdn
 0  0  974.67        search event
 0  0  975.00        move   event
```

The first part of the event history is a print-out of the in-view out-of-view segment lengths in meters. This is followed by the event history itself. Each line contains the number of the active combatant, the target, perhaps the time, the event name or a clue to what is happening, and the name of the subroutine that calls for a print of the event by the **history** routine. Times are printed only when **events** calls for the print out. Blank times occur at the same time and are associated with the event time shown just before in the output. The blank space after the time is reserved to print out times associated with scheduling and cancelling events.

2.3.4 Monitoring a Single System Force Wars is able to display the history of a single system so that a programmer can debug the model and a user can learn how the program works. Hopefully, no debugging will be required unless the program is changed. Here, we discuss how to produce the single system history and describe the information printed.

Producing the single system history. Change the second line of the **d.kinds** file. Set the first value on the line to 1 so that only one replication is simulated. Set the value on the second line to the ID of the system. If you are interested in a replication other than the first, set the third value to the random number seed used at the beginning of that replication. For example, if you wish to see results for system 5, the second line will look something like this:

```
1 5 0 1 1111111 999.0      # of reps, firer, target, and rep to monitor, random seed, tmax
```

The Output for a Single System. The output will contain the input echo, the event history of a single system, and a summary of the replication. The first and last of these have been discussed earlier. The event history contains one line for each event. The line contains the following information:

2.3.5 Error messages. The error messages are:

1. Fired_burst: Not implemented. Firer is 3
2. Impact: tgt in full defilade. Tgt j= 3
3. Move: no data for: T81 3
4. Search: L must be 1..7. L = 9

You should not be able to get the first message. If you do, contact the author. If you get the second message, there is a bug in the program, so contact the author. If you get the third message, the **d.move** file indicates there is an actual combatant for which there are no data in the **d.kinds** file.

If you get the fourth error message, it means the distance between the target and the firer is more than 3 km. Contact the author.

3. DATA STRUCTURE

This section discusses the global data, the data that will be used by more than one routine. It also discusses the input data you will need to run the program and how it must be organized. The key files are:

ITEM	COMMENT
globals.h	Contains common statements for global variables.
input.f	The routine that reads miscellaneous data.
rdrow.f	Reads a row of data into the main data table.
move.f	Reads initial deployment and motion data.
search.f	Reads acquisition data.

The major classes of data are:

1. Control data.
2. Parameters and constants.
3. Data describing kinds of systems.
4. Data for as many as 50 combatants.
5. Sensing data for as many as 10 kinds of systems versus as many as 10 targets.
6. Kill probabilities for multiple ranges, types of rounds, firers, and targets.
7. Relationship data for firer versus target.

Control Data. These data control how many replications are performed, how long they last, and how much output is generated. Here are the definitions of the variables.

ITEM	COMMENT
lev	For future use as print control.
i_m	# of firer or other active system to monitor. Varies from 1 to i_s
i_s	# of weapon systems. Varies from 2 to *IX*.
j_m	# of target to monitor. Varies from 1 to i_s
k_s	# of kinds of systems being played. Varies from 1 to *KX*.
κ	Stores statistics regarding changes in connections.
n_m	# of battle to monitor.
n_n	# of current battle. ($1 \leq n_n \leq n_s$)
n_s	# of battles to simulate.
search_on	True if and only if search is scheduled to be called.
$t_{\max}$	Maximum time for a battle to continue (s).
trace	Used to control debug prints.

Parameters and other constants. These are named to avoid mysterious magic numbers in the code. Here are the definitions of the constants:

ITEM	COMMENT
	Parameters:
IX	Maximum number of systems (entities) in game.

KX	Maximum number of kinds of systems in game.
MX	Maximum number of items in table.
	Constants:
ALLx	= 0 (used when all possible targets are specified)
NULL	= 0 (used when no target is specified)
	Exposure constants:
FD	= 1 (Full defilade)
TD	= 2 (Turret defilade)
HD	= 3 (Hull defilade)
FE	= 4 (Fully exposed)
	Values for λ_{ij}:
ISDEAD	= 0 (Can be ignored. Is dead or in hiding.)
ISOUT	= 1 (Is out of detection range.)
INSIDE	= 2 (Is inside detection range but not in LOS.)
INLOS	= 3 (Is inside detection range and in LOS.)
ISSEEN	= 4 (Is detected.)
ISFREN	= 5 (Is classified as a friend, correctly or incorrectly.)
ISFOE	= 6 (Is classified as a foe, correctly or incorrectly.)
ISTGT	= 7 (Is chosen for engagement.)

Data describing kinds of systems. Force Wars stores general data, common to all systems of a given type, in the *itbl* array as integer values. The *atbl* array stores real values and exactly overlaps the *itbl* array. These arrays are dimensioned with MX rows and IX columns (or vice versa if you think in Fortran instead of normal scientific language.) Currently this allows 50 rows and 10 columns, hence 10 types of weapon systems.

The variables used to store the data are defined as follows:

ITEM	COMMENT
itbl	Stores a description of each kind of system as integers.
atbl	Uses same storage as *itbl* but stores reals.

Here is an example of the data stored in the table. In this case, data for 8 of a possible 10 kinds of systems are read in.

```
M60     M1      T72     T80     BFV     Apache          Hind    Inf
3       3       3       3       2       1       1       2       # weapons
55      39      40      39      30      30      30      5000    Wpn 1 ammo
200     200     200     200     200     0       0       200     Wpn 2 ammo
5000    5000    5000    5000    0       0       0       0       Wpn 3 ammo
10.     10.     10.     10.     10.     10.     10.     10.     tbr
2       2       2       1       2       2       2       2       # guidance channels for wpn 1
1.      1.      1.      1.      1.      1.      1.      1.      vm - 1
1.      1.      1.      1.      1.      1.      1.      1.      vm - 2
1.      1.      1.      1.      1.      1.      1.      1.      vm - 3
```

```
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   beta - 1
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   beta - 2
0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   beta -3
9.9   9.9   9.9   9.9   9.9   9.9   9.9   9.9   tfirst
0.3   0.3   0.3   0.3   0.3   0.3   0.3   0.3   pinpnt?
1     1     1     1     1     1     1     1     loader type?
2     2     999   999   999   999   999   999   Rounds shot until popdown
2     2     3     3     3     3     3     3     Rounds shot until switch
```

Data describing individual combatants. Individual combatants are described by data stored in arrays. The array elements and their definitions follow:

ITEM	COMMENT
$alive_i$	True if and only if system i is alive.
e_{i4}	Exposure of combatant i in one of four categories.
g_i	Time system i halted (s).
h_i	Cumulative time delay of system i along path (s).
s_{ik}	The kth coordinate of the ith system (km).
$kind_i$	Kind of system for system i.
λ_{ij}	Connection between system i and j.
$mision_i$	'DEFEND', 'ATTACK' are only values used now.
$moving_i$	True if and only if system i is in motion.
$name_i$	Contains the name of a kind of system, e.g., 'M1A1'.
n_r	
$narmy_i$	Side system i is on (1-2).
$nbuddy_i$	# of system that is i's buddy for pop-up/pop-down and bounding overwatch.
nrf_i	# of rounds i fired during current engagement of this target.
nrd_i	
nrf_i	# of rounds i fired during current engagement of this target.
$nwpn_i$	Kind of round or weapon (1-3).
$nchan_i$	# of busy guidance channels for combatant i.
$owatch_i$	True if and only if attacker with buddy is in overwatch. False if bounding.
u_i	Maximum speed of the ith system (m/s).
v_{ik}	Combat velocity of ith system (m/s).

Sensing data. Sense reads sensing information into the *tbl* array. It contains data for as many as 10 kinds of systems sensing as many as 10 kinds of targets at as many as 8 ranges. This is used to randomly generate detection distances for each combatant, which is stored in the *d* array.

ARRAY	COMMENT
d_{ik_jm}	Detection range for system i versus target k_j in posture m (m).
n_r	Number of ranges for which data are tabled. (WARNING: this is clobbered when pk's are read.!)
$tbl_{nmk_jk_i}$	Detection distance at range n for posture m for searcher k_i and target k_j (m).

Kill probabilities. Input reads these values into the **p** array.

ARRAY	COMMENT
$p_{lkk_jk_i}$	Probability of kill for kind of firer k_i kind of target k_j kind of round k at range l.

Relationship data. Relationships between the combatants and their targets are stored in several arrays. They are defined below:

ARRAY	COMMENT
λ_{ij}	Connection between system i and j.
r_{ij}	Distance between i and j (km)

3.1 Globals.h: The Global Variables

First, let's discuss the global variables. We wish to choose names for them wisely and minimize the number we must remember. The names should be familiar and easy to remember if possible. The ones that will occur in equations should be compact so the equations will appear as normal mathematical equations. This means they are limited to a single character possibly with some subscripts. We consider a spelled greek letter to be a single character. Those not appearing in equations can be longer.

```
integer IX, KX, MX ! # entities, # kinds of entities, #items for each in table.
parameter (IX=50, KX=10, MX=50)
character*6 trace, mision*6, name
integer ALLx, NULL, FD, TD, HD, FE
integer ISDEAD, ISOUT, INSIDE, INLOS, ISSEEN, ISFREN, ISFOE, ISTGT
logical alive, search_on, owatch, busy, moving
integer lev, is, im, jm, ks, nm, nn, ns
integer nrd, nrf, nwpn, narmy, nbuddy
integer kappa, lambda, itbl, nr
integer nchan, kind, e
real tmax, g, h, s, u, v, r, p, tbl, d

common /chars/ trace, mision(IX), name(KX)
common /consts/ ALLx, NULL, FD, TD, HD, FE
common /const2/ ISDEAD, ISOUT, INSIDE, INLOS, ISSEEN, ISFREN, ISFOE, ISTGT
common /contrl/ lev, is, im, jm, ks, nn, ns, nm, search_on, tmax
common /ix01/ alive(IX), g(IX), h(IX), kind(IX), moving(IX), nchan(IX)
common /ix02/ nrd(IX,3), nrf(IX), nwpn(IX), narmy(IX), owatch(IX), busy(IX)
common /ix03/ e(IX,4), s(IX,3), u(IX), v(IX,2)
common /ix04/ nbuddy(IX)
common /ixix/ kappa(IX,IX,15), lambda(IX,IX), r(IX,IX)
common /kx01/ itbl(MX,KX)
common /kxkx/ nr, p(10,3,KX,KX), tbl(10,8,KX,KX)
common /sens3/ d(IX,KX,3)

real atbl(MX,KX)
equivalence (itbl(1,1),atbl(1,1))
```

/chars/	Contains character variables.
/consts/	Contains constants.
/const2/	Contains more constants.
/contrl/	Contains variables that control execution of the run.
/ix01/	Contains variables for each system.
/ix02/	Contains variables for each system.
/ix03/	Contains variables for each system.
/ix04/	Contains variables for each system.
/ixix/	Contains arrays for system i and target j.
/kx01/	Contains MX values for as many as KX kinds of systems.
/kxkx/	Contains values for the KX kinds of systems and the KX kinds of targets.
/sens3/	Contains data for IX systems and their KX kinds of targets.

3.2 Other Common Statements

A few other routines use common statements that do not appear in the global.h file. **Appear, bound, vanish,** and **terain** use the following common:

```
common /terane/ del(100), a(IX,3), q(IX), iseg(IX)
```

5. Sensing data for as many as 10 kinds of systems versus as many as 10 targets.

ITEM	COMMENT
∇_i	In view segment length (m) for i odd.
	Out of view segment length (m) for i even.
a_{ij}	Jth coordinate of system i, when it last mask or unmask (m).
q_j	Segment length system j must travel to mask or unmask (m).
$iseg_i$	# of segment in ∇ that moving system is in.

The clock routines, **reset, skedul, cancel,** and **event** use these labeled commons:

```
common /event1/ what(NE)
common /event2/ when(NE), who(NE), whom(NE), next(NE), nxevnt, nxidle, prflag
```

ITEM	COMMENT
$what_k$	Name of kth event.
$when_k$	Time of kth event (s).
who_k	Combatant performing kth event.
$next_k$	Link to next event after this one.
nxevnt	Link to first event.
nxidle	Link to first idle event node.
prflag	True if and only if printing desired.

The routines **ranu, input** and **stats** use the random number seed in the following common:

```
common /crandm/ jseed
```

The routines **ckdis, diseng,** and **blkdat** use the following common:

```
common /debug/ mu(8,8)
```

A final common appears only in **terain**. It stores intervisibility data which will eventually be read in.

```
common /terrac/ a1, b1, a2, b2
```

ITEM	COMMENT
a_1	Weibull shape factor for in-view segment length distribution.
a_2	Weibull shape factor for in-view segment length distribution.
b_1	Weibull shape factor for in-view segment length distribution.
b_2	Weibull shape factor for in-view segment length distribution.

3.3 Variables in Alphabetic Order

Here is a list of constants and variables in alphabetical order.

IX	Maximum number of systems (entities) in game.
KX	Maximum number of kinds of systems in game.
MX	Maximum number of items in table.
$alive_i$	True if and only if system i is alive.
d	
e_{ik}	Exposure of ith system due to kth condition.
g_i	Time system i halted (s).
h_i	Cumulative time delay of system i along path (s).
i_m	# of firer or other active system to monitor. Varies from 1 to i_s
i_s	# of weapon systems. Varies from 2 to IX.
$itbl_{jk}$	jth value for kth type of system. See input.
j_m	# of target to monitor. Varies from 1 to i_s
k_s	# of kinds of systems being played. Varies from 1 to KX.
κ_{ijk}	Count of times relationship k occured for firer i versus target j.

k	Relationship
1	j started within or entered i's detection range.
2	j exited i's detection range.
3	i detected j while it was not firing.
4	i detected j by its firing signature.
6	i classified j as friend.
7	i classified j as foe.
8	i engaged j.
9	i fired at j.
10	i killed j
11-13	Unused
14	i made first killing shot on j.

$kind_i$	Kind of system for system i.
λ_{ij}	Connection between system i and j.
$mision$	'DEFEND', 'ATTACK' are only values used now.
$moving_i$	True if and only if system i is in motion.
n_m	# of battle (replication) to monitor.
n_n	# of current battle. ($1 \leq n_n \leq n_s$)
n_r	# of ranges for which data is tabled.
n_s	# of battles to simulate.
$narmy_i$	Side system i is on (1-2).
$nbuddy_i$	# of system that is i's buddy for pop-up/pop-down and bounding overwatch.

$nchan_i$	# of busy guidance channels for i.
nrd_{ik}	# of rounds of ammunition remaining for weapon k on system i.
nrf_i	# of rounds i fired during current engagement of this target.
$nwpn_i$	Kind of round or weapon (1-3).
$owatch_i$	True if i is in overwatch. False otherwise.
p	Probability of kill table.
r_{ij}	Distance between i and j (km)
s_{ik}	The kth coordinate of the ith system (km).
$t_{\max}$	Maximum time for a battle to continue (s).
tbl_{jk}	jth value for kth type of system. See input.
trace	Used to control debug prints.
u_i	Maximum speed of the ith system (m/s).
v_{ik}	Combat velocity of ith system (m/s).
ALLx	= 0 (used when all possible targets are specified)
FD	= 1 (Full defilade)
FE	= 4 (Fully exposed)
HD	= 3 (Hull defilade)
INLOS	= 3 (Is inside detection range and in LOS.)
INSIDE	= 2 (Is inside detection range but not in LOS.)
ISDEAD	= 0 (Can be ignored. Is dead or in hiding.)
ISFOE	= 6 (Is classified as a foe, correctly or incorrectly.)
ISFREN	= 5 (Is classified as a friend, correctly or incorrectly.)
ISOUT	= 1 (Is out of detection range.)
ISSEEN	= 4 (Is detected.)
ISTGT	= 7 (Is chosen for engagement.)
NULL	= 0 (used when no target is specified)
TD	= 2 (Turret defilade)
search_on	True if and only if search is scheduled to be called.

4. TOP LEVEL ROUTINES

4.1 Hierarchy of the Routines

The hierarchy diagram below shows the relationship of the routines. Each routine is indented under the routine that calls it.

```
main
    input
        rdrow
        move
        sense
    reset
    init2
    move
    terain
        bound
    events
        event
        history
        move
        search
            clasif
            select
                selec2
        detect_rg
        select
        fire
            pinpnt
            fired_missile
                call diseng
                call select
            fired_single
                call diseng
            fired_burst
        impact
            fired_missile
            damage
                call diseng
        bound
        popup
        popdn
            diseng
        appear
            aprter
                los
        vanish
            vanter
                diseng
        show
stats1
```

4.2 Main: The Main Program

Main and its subsidiary routines simulate combat between a mix of weapon systems on each side. **Main** calls these routines to do these tasks:

input	Read data describing combatants, etc
reset	Reset the clock and event calendar at the beginning of battle.
init2	Initialize values at the beginning of a battle.
move	Start combatants moving.
terain	Start terrain mask/unmask sequences.
skedul	Schedule **detect_rg** to start checking when contact is made.
events	Call routine to handle event as it comes up.
stats2	Accumulate statistics for a single battle.
stats1	Accumulate and print statistics for the entire run.
ckdis	DEBUG to check disengagement in λ matrix.

The DO loop controls the execution of n_s battles (replications).

The main routine is quite short so that it may be modified. Usually, a programmer would modify the routine by inserting a DO loop that would override a parameter and run the program using a set of values for the parameter of interest.

Key globals:

κ_{ijk}	Summary statistics array for ith firer, jth target, and kth datum.
n_s	No. of battles to simulate.
n_n	No. of current battle.

```
PROGRAM FWARS
!Simulate force on force combat. 2 Sep 93
include 'global.h'

call input
kappa = 0
DO nn=1,ns
  write(3,*) 'Replication', nn
  call reset (trace == 'clock ' .and. nm == nn)
! call reset (nm == nn)
  call init2
  call move (0.0,1)
  call terain
  call skedul(0, 0, 'det_rg', 0, 'main  ')
  call events (tmax)
  call stats2
ENDDO
call stats1
call ckdis
END
```

4.3 Input: Read Data for Each Kind of System

Data read:

trace	Character string governing debug prints.
n_s	No. of battles to simulate.
i_m	Active combatant to monitor.
j_m	Target to monitor.
n_m	Battle to monitor.
j_{seed}	Random number seed.
$t_{\max}$	Maximum time for any battle to run (s).
k_s	No. of kinds of systems to read data for.
$name_k$	Name of kth system.

Many of the input data are read into a table. Each row of data is read by calling **rdrow**, which also echos the data if desired. The data read into the table are as follows:

ROW	TYPE OF DATA
1	No. of combatants of each kind.
2	No. of weapons on this kind of system.
3	No. of rounds for Weapon 1.
4	No. of rounds for Weapon 2.
5	No. of rounds for Weapon 3.
6	Time between rounds for Weapon 1.
7	No. of guidance channels for Weapon 1.
8	Muzzle velocity for Round 1 (km/s).
9	Muzzle velocity for Round 2 (km/s).
10	Muzzle velocity for Round 3 (km/s).
11	Siacci β for Round 1 (hz).
12	Siacci β for Round 2 (hz).
13	Siacci β for Round 3 (hz).
14	Time to launch first round (s).
15	Probability of pinpoint detection.
16	Loader type (1=manual, 2=load assist, 3=auto loader)
17	Rounds to fire before defender pops down.
18	Rounds to fire before switching targets.

After the table is filled, **move** is called to read in the initial deployment of the combatants and **sense** is called to read in the sensor characteristics for each type of system. Finally, **input** reads the probability of kill data into the array p_{lkji}, where:

Locals:

i	Kind of firer.
j	Kind of target.
k	Round (or weapon) # (1-3).

l No. of ranges for which data are tabled.

```
 SUBROUTINE INPUT
 include 'global.h'
 common /crandm/ jseed
1 format(a16,8a8)
2 format(a16,8i8)
4 format(8a6)

 if (trace == '><') write(3,*) '>input'
 open(3,file="output",status='new')
 rewind(3)
 write(3,*)
 open(4,file="d.kinds",status='old')
 rewind(4)
 read(4,4) trace
 write(3,1) 'INPUT ECHO:      '
 read(4,*) ns, im, jm, nm, jseed, tmax
 write(3,2) 'Replications    ', ns
 ! Read description for each kind of system.
    read(4,*) ks
    write(3,2) '# kinds of combatants read', ks
    read(4,*) (name(k), k=1,ks)
    write(3,1) 'Name of kind    ', (name(k), k=1,ks)
 !  call rdrow(1, ks, 1, '# of each kind  ')
    call rdrow(2, ks, 1, '# of weapons    ')
    call rdrow(3, ks, 1, 'Weapon 1 ammo   ')
    call rdrow(4, ks, 1, 'Weapon 2 ammo   ')
    call rdrow(5, ks, 1, 'Weapon 3 ammo   ')
    call rdrow(6, ks, 2, 'Time between rds')
    call rdrow(7, ks, 1, 'Guidance channel')
    call rdrow(8, ks, 2, 'Muz vel 1 (km/s)')
    call rdrow(9, ks, 2, 'Muz vel 2 (km/s)')
    call rdrow(10,ks, 2, 'Muz vel 3 (km/s)')
    call rdrow(11,ks, 2, 'Wpn 1 beta (hz) ')
    call rdrow(12,ks, 2, 'Wpn 2 beta (hz) ')
    call rdrow(13,ks, 2, 'Wpn 3 beta (hz) ')
    call rdrow(14,ks, 2, 'Tfirst           ')
    call rdrow(15,ks, 2, 'Pinpnt           ')
    call rdrow(16,ks, 1, 'Loader type      ')
    call rdrow(17,ks, 1, 'Rds to pop-down ')
    call rdrow(18,ks, 1, 'Rds to disengage')
    close(4)
 call move(0.0,0) ! Read in initial deployment
 call sense(0)    ! Read sensing data
 ! Read lethality data
    open (4,file='d.kill',status='old')
    rewind 4
    read(4,*) nr
    DO i=1,ks
      DO j=1,ks
        DO k=1,3
           read(4,*) (p(l,k,j,i),l=1,nr)
        END DO
      END DO
    END DO
    close (4)
    write(3,*)
  if (trace == '><') write(3,*) '<input'
END
```

4.4 Rd_Row: Read and Echo One Datum for Each Kind of System

Rdrow simply reads a row of numbers into the table *itbl* as integers or the table *tbl* as reals. The tables are actually one table that goes by two names so that integers and reals may be stored in it. Each column in the table contains data for a single kind of system, for example M1A1's.

Arguments:

n_{row}	No. of the row.
n_{col}	No. of columns.
k	1 if and only if read integers, 2 if read reals.
str	Description of data on row.

Key globals:

itbl	Matrix of integers.
tbl	Matrix of reals.

```
SUBROUTINE RDROW(nrow, ncol, k, str)
!Read a row of miscellaneous information.
include 'global.h'
character str*16
logical echo
1 format(a16,8i8)
2 format(a16,8f8.2)

if (trace == '><') write(3,*) '>rdrow'
echo = .true.
IF (k == 1) THEN! Read row of integers
  read(4,*) (itbl(nrow, n),n=1,ncol)
  if (echo) write(3,1) str, (itbl(nrow,n),n=1,ncol)
ELSE! Read row of reals
  read(4,*) (atbl(nrow, n),n=1,ncol)
  if (echo) write(3,2) str, (atbl(nrow,n),n=1,ncol)
ENDIF
if (trace == '><') write(3,*) '<rdrow'
END
```

4.5 Events: Simulate Each Event in the Battle

Events loops through each event in the battle until the 'end_sim' event occurs or the simulation time is greater than $t_{\max}$. It calls **event** to find the next event and then branches to that event and loops back for the next one.

Argument:

t_{mazz}	Maximum time for battle (s).

Locals:

time	Simulation time (s)
who	Firer, searcher, or mover.
what	The next event to simulate.

whom Target (if specified, zero otherwise).

```
SUBROUTINE EVENTS (TMAXX)
!Loop thru events in battle.
INCLUDE 'global.h'
integer who, whom
character what*6

PRINT*,'Events: nn, nm=', nn, nm
DO WHILE (time < TMAXX)
  call event(time,who,what,whom)
  call history (who, whom, time, what, 'events')
  IF (what == 'move  ') THEN
    call move(time,2)
  ELSEIF (what == 'search') THEN
    call search(time)
  ELSEIF (what == 'det_rg') THEN
    call detect_rg(time)
  ELSEIF (what == 'select') THEN
    call select(time,who)
  ELSEIF (what == 'fire  ') THEN
    call fire(time,who,whom)
  ELSEIF (what == 'impact') THEN
    call impact(time,who,whom)
  ELSEIF (what == 'bound ') THEN
    call bound (who,1,time)
  ELSEIF (what == 'popup ') THEN
    call popup (time,who,whom)
  ELSEIF (what == 'popdn ') THEN
    call popdn (time,who)
  ELSEIF (what == 'appear') THEN
    call appear (time,who)
  ELSEIF (what == 'vanish') THEN
    call vanish (time,who)
  ELSEIF (what == 'endsim') THEN
    time = TMAXX
  ELSE
    write(3,*) 'EVENTS: no such event as ', what
    STOP
  ENDIF
  if (ns == 1) call show (time)
END DO
END
```

4.6 Init2: Set values at the beginning of a replication

Init2 initializes system values at the beginning of each replication.

Key globals:

$alive_i = .true.$ Initially, system i is alive.

$e = FE$ At game start, all systems tentatively set to Fully Exposed.

k=$nbuddy_i$ ID of buddy of system i.

$nchan_i = 0$ System i has no guidance channels in use.

$nwpn_i = 0$ System i has selected no weapon.

busy System i is not too busy to select a target.

nrd_{ij} No. Rds available for weapon j on system i

nrf_i # rounds fired during current pop-up.

Posture of systems. At the beginning of each replication, INIT2 sets the exposure of each system. Defending tanks are all in turret defilade (TD). Generally, attacking tanks are fully exposed (FE) but if bounding overwatch tactics are used, the tanks are grouped in pairs and one of each pair is advancing and fully exposed while the other is in turret defilade. The pairing is defined by the vector *nbuddy*, where $nbuddy_i$ is the ID of the buddy of system i.

If an attacking system has a buddy, the posture of both is set at the same time. The vector **e** is checked to assure the posture of the second is not set twice.

```
SUBROUTINE INIT2
!Set values for beginning of replication.
include 'global.h'

if (trace == '><') write(3,*)'>init2'
alive = .true.
e = FE
nchan = 0
nrf = 0
nwpn = 0
search_on = .false.
call sense(1)
DO i=1,is
   nrd(i,1) = itbl(3,kind(i))
   nrd(i,2) = itbl(4,kind(i))
   nrd(i,3) = itbl(5,kind(i))
!  Initialize posture (exposure) of each system.
      if (mision(i) == 'Defend')  e(i,1:2) = TD
      IF (mision(i) == 'Attack')  THEN
!     Set exposure of systems with buddies.
         k = nbuddy(i)
         if (k > 0 .and. i > k) call bound (i, 0, 0.0)
      ENDIF
   kappa(i,1:is,13) = 0
END DO
lambda = 1
if (trace == '><') write(3,*)'<init2'
END
```

5. ACQUISITION ROUTINES

Detect_rg, **sense**, **search**, **classify**, and **pinpnt** simulate acquisition of targets. **Sense** reads sensing data at the beginning of the run and draws maximum detection ranges for each system at the beginning of each battle (replication). **Search** is called every second while undetected targets are in detection range and in line of sight for any system. It randomly determines whether those targets are detected. **Pinpnt** is called by **fire** each time a round is fired to see if any systems have the firer in detection range and line of sight. If so, it randomly finds whether those systems detect the firing signature. When either of these detection routines finds that a target is detected, it calls **clasif** to simulate classification of the target as friend or foe.

The global data used by these routines are stored in the arrays *tbl* and *d*. *Tbl* has four subscripts, with the third identifying the kind of target and the fourth identifying the kind of searcher. If the third and fourth subscripts are given values, they identify a matrix. For example, if the target is of kind 3 and the firer is of kind 4, we have $tbl_{n,m,3,4}$ which looks as follows:

```
7 M60 M60
Rg   S-HD S-FE M-FE   S-HD   S-FE   M-FE Pfoe
0.0 0.99 1.00 1.00 0.014 0.019 0.018 0.00
0.5 0.70 0.90 0.95 0.008 0.015 0.015 0.01
1.0 0.40 0.82 0.90 0.003 0.009 0.012 0.02
1.5 0.25 0.65 0.80 0.002 0.007 0.009 0.04
2.0 0.15 0.45 0.60 0.002 0.005 0.006 0.05
2.5 0.05 0.15 0.20 0.001 0.002 0.003 0.06
3.0 0.00 0.00 0.00 0.000 0.000 0.000 0.07
```

When the data are read, the values shown in the rows are placed in successive memory positions. This makes it easy for the program to interpolate on range. The column headings are:

s	stationary target
m	moving target
hd	hull defilade target
fe	fully exposed target

The first column contains range (km), the following columns contain probabilities. They are:

Col	Value
2-4	Probability that target is inside detection range.
5-7	Probability of detecting in the next second given target is in detection range.
8	Probability of classifying as a foe.

At the beginning of each battle, **sense** randomly draws the detection range for each system. For System 1 and target kind 3, it draws a random number and

interpolates in Columns 2 through 4 of the table above to find the maximum detection range for each of the three conditions: stationary, hull defilade target; stationary, fully exposed target; and moving, fully exposed target. These are stored in $d_{1,3,1}$, $d_{1,3,2}$, and $d_{1,3,3}$.

5.1 Detect_rg: Find When a Target Moves Into Detection Range

Every 5 seconds, the program calls **detect_rg** which checks to see if any combatants have entered or exited the detection range of any other combatants. At time zero, it checks to see if any are already within range. Normally, a combatant will have some friendlies within range. Later, foes will come into range and perhaps exit.

The routine loops through all firers and all targets and considers each live pair. Each firer has three detection ranges associated with it for each kind of target. The three detection ranges are for stationary, hull defilade targets; stationary, fully exposed targets; and moving, fully exposed targets. Because each kind of target may differ in size and contrast, they may differ in the range to which they can be detected, so each kind has a set of detection ranges associated with it.

If a target j has just entered a firer i's detection range, **detect_rg** sets $\lambda_{ij} = INSIDE$ and calls **los** to find if line of sight exists between i and j. If a target j has just exited a firer i's detection range, **detect_rg** calls **diseng** to reduce the connection between them back to $\lambda_{ij} = ALIVE$. **Diseng** also handles other chores such as disengaging i from j and having i select a new target, aborting missiles i may have fired at j, and so on.

Note that if j is fully exposed and within the associated detection range, it may exit detection range when it switches to an overwatch role and consequently assumes a hull defilade posture. Any change of exposure may result in the appearance or disappearance of the target from detectability.

After **detect_rg** has done this for all pairs of combatants, it reschedules itself to be called in 5 seconds.

Argument:

t_s	Simulation time (sec).

Key globals:

r_{ij}	Distance from i to j (km).
i_s	Number of combatants.
$alive_j$	True if and only if j is alive.
$kind_j$	Kind of system j is.
e_{j1}	Exposure of j. = 1 if full defilade, 2 if turret defilade, etc
λ_{ij}	Relationship of firer i and target j. E.g. = 1 if alive and out of range, = 7 if i has selected j to engage.
κ_{ij1}	Count of times j entered i's detection range.
κ_{ij2}	Count of times j exited i's detection range.

d_{ikm} Maximum detection range of ith combatant against target type k for target in posture m (m).

Locals:

i ID of searcher.

j ID of target.

m = 1 if stationary hull defilade target, = 2 if stationary fully exposed target, = 3 if moving fully exposed target.

```
SUBROUTINE DETECT_RG (ts)
!Find targets that have moved into or out of detection range.
include 'global.h'
integer m ! Exposure, motion index 1=HD-stationary, 2=FE-stationary, 3=FE-moving
save
1 format(20i4,(4x,19i4))
2 format(i3,' lambda=',20i3)
3 format(a,3i3,2f6.2)

DO i=1,is
  IF (alive(i)) THEN! Find who i can detect.
    DO j=1,is
      k = kind(j)
      m = 3          ! Tentatively set for FE-moving tgt.
      if (.not. moving(j)) m = 2! Reset for FE-stationary tgt.
      if (e(j,1) < FE) m = 1! Reset for HD-stationary tgt.
      if (trace=='det_rg') write(3,3)'Detect_rg: i,j,m,r,d=',i,j,m,r(i,j),d(i,k,m)
      IF ((i==j) .or. .not.alive(j)) THEN! Do nothing. Won't detect self or dead tgt.
      ELSEIF (r(i,j)< d(i,k,m)) THEN! Find if j just entered i's det rg.
        IF (lambda(i,j) < INSIDE) THEN! Mark i to search for j because j just entered.
          call history(i, j, ts, 'enter ', 'det_rg  ')
          lambda(i,j) = INSIDE
          kappa(i,j,1) = kappa(i,j,1) + 1
          call los(ts,i,j)
        ENDIF

      ELSEIF (lambda(i,j) >=INSIDE) THEN! Break off engagement. Tgt j just left.
   call history (i, j, ts, 'exit  ', 'det_rg  ')
        kappa(i,j,2) = kappa(i,j,2) + 1
        call diseng (ts,i,i,j,ISOUT)
      ELSE
      ENDIF
    END DO
    if (trace=='det_rg') write(3,2) i, (lambda(i,j), j=1,is)
  ELSE
    if (trace=='det_rg') write(3,*) 'Detect_rg: alive=',(alive(m),m=1,is)
  ENDIF
END DO

call skedul(ts+5.0,0,'det_rg',0, 'det_rg')
IF (trace=='det_rg') THEN
  write(3,*)'i   Status of i''s knowledge of j'
  DO i=1,is
     write(3,*) i, (lambda(i,j),j=1,is)
  END DO
ENDIF
END
! for lambda(i,i) 1 = i doesn't search,  3 = i searches (unseen tgts in LOS.)??
! for lambda(i,j) 0 = dead, 1 = ou rg,  2 = in rg,  3 = in LOS,
!                 4 = seen, 5 = friend,  6 = foe,   7 = engaged
```

5.2 Sense: Read Sensor Data and Find Detection Ranges

Sense is called with a zero argument at the beginning of each run. When this happens, **sense** reads in a table of sensing data for each kind of system against every other kind of system.

Sense is called with a non-zero argument at the beginning of each battle (replication). When this happens, it randomly finds the detection range for each system i against each kind of system k_j and for three conditions. The conditions are 1) stationary searcher versus stationary target in hull defilade, 2) stationary searcher versus stationary, fully exposed target, and 3) stationary searcher versus moving, fully exposed target.

Argument:

new	Zero if and only if called at beginning of replication by **input**.

Locals:

k_i, k_j	Kind of system for i, j.
n_r	# of ranges in table (rows).
tbl_{n,m,k_jk_i}	Sensing probability table.
	$m = 1$ range column.
	$m = 2$ probability of ever detecting stationary, hull defilade target.
	$m = 3$ probability of ever detecting stationary, fully exposed target.
	$m = 4$ probability of ever detecting moving, fully exposed target.
	$m = 5$ probability of detecting stationary, hull defilade target in next second.
	$m = 5$ probability of detecting stationary, fully exposed target in next second.
	$m = 7$ probability of detecting moving, fully exposed target in next second.
	$m = 8$ probability of classifying as foe.
q	Random draw from uniform distribution.
n	Range index.
$d_{i,kj,m}$	Distance within which system i can see systems of kind kj under condition m.
m	1 = SS HD, 2 = SS FE, 3 = SM FE

```
SUBROUTINE SENSE(new)
! Read sensing data and find detection ranges.
implicit none
integer i, ki, kj, m, n, new
real f, q, ranu
include 'global.h'
save
2 format(f8.1,7f6.2)
3 format(i2,1x,a5,12f6.2, (8x,12f6.2))
f(n,m) = tbl(n,1,kj,ki) + &
    (q-tbl(n,m,kj,ki))*(tbl(n+1,1,kj,ki)-tbl(n,1,kj,ki)) / &
    (tbl(n+1,m,kj,ki)-tbl(n,m,kj,ki))

if (trace == '><') write(3,*) '>sense: m=', m
IF (new == 0) THEN! Read sense data at beginning of run.
   open(2,file='d.sense',status='old')
   rewind 2
```

```
      read(2,*)
      DO ki=1,ks
         DO kj=1,ks
            read(2,*) nr
            read(2,*)
            read(2,*) ((tbl(n,m,kj,ki),m=1,8),n=1,nr)
         END DO
      END DO
      close (2)
      IF (trace == 'sense ') THEN
         write(3,*) 'Range (km), 3 Pinf, 3 P1sec, Pfoe for sys1 vs sys1'
         write(3,2) ((tbl(n,m,1,1),m=1,8),n=1,nr)
      ENDIF

   ELSE! Find detection ranges at beginning of replication.
      if (trace == 'sense ') write(3,*) 'DETECTION RANGES (m)'
      DO i=1,is
         q = ranu()
         DO kj=1,ks! Find detection range for i detecting type L.
             ki = kind(i)
             DO m = 1,3
               call hunt(tbl(1,m+1,kj,ki),7,q,n)
               IF (n < 1) THEN
                   d(i,kj,m) = tbl(1,m,kj,ki)
               ELSEIF (n < 7) THEN
                  d(i,kj,m) = f(n,m+1)
               ELSE
                  d(i,kj,m) = tbl(7,m,kj,ki)
               ENDIF
             END DO
          END DO
          if (trace == 'sense ') write(3,*) 'Sense: random, N=', q, N
          if (trace == 'sense ') write(3,3) i, 'SS-HD', (d(i,kj,1), kj=1,ks)
          if (trace == 'sense ') write(3,3) i, 'SS-FE', (d(i,kj,2), kj=1,ks)
          if (trace == 'sense ') write(3,3) i, 'SM-FE', (d(i,kj,3), kj=1,ks)
      END DO
   ENDIF
   if (trace == '><') write(3,*)'<sense'
   ! i - id of firer.
   ! j - unused.
   ! ki - kind of firer.
   ! kj - kind of target
   ! m - column index in data table.
   ! n - row (range) index.
   ! new - 0 iff time to read data at beginning of run. Non zero if at beginning
   !  of replication and time to pick random detection ranges.
   !
   END
```

5.3 Search: Simulate Search for 1 Second

Search finds which systems detect other systems in the next second. It loops through all searchers, checking whether they are alive and whether are systems in their line of sight. If so, it loops through all other systems that are alive, checking whether they are detected. If a system j is in line of sight of system i, **search** finds the range to j and the probability i will detect it. **search** then draws to find if i detects j. If so, it scores a detect and calls **clasif** to classify j as a friend or foe. If not, it records that i still has an undetected target in line of sight and that **search** must reschedule itself. Finally, after all searchers and targets are considered, **search** reschedules itself in one second, as necessary.

Fully exposed systems may detect targets, but if they are the bounding partner in a bounding / overwatch pair, they do not fire. The variable can_fire controls whether i can fire. If i is not fully exposed or if i has no buddy, then it is able to fire.

Arguments:

t_s	Simulation time (sec).

Locals:

L	Interpolate between rows L, L+1.
can_fire	True if and only if i is less than fully exposed or has no buddy.
i	ID of searcher.
j	ID of potential target.
k_i	Kind of system for system i.
k_j	Kind of system for system j.
m	Interpolate in column m.
p_f	Probability that i detects j in the next second.
r_{ij}	Range from i to j.
see_foe	True if and only if i sees a target and classifies it as a foe.
seen	True if and only if i detects j.
y_1	
y_2	

```
SUBROUTINE SEARCH(ts)
!Find who detects whom in the next second. 7 Apr 95
include 'global.h'
logical can_fire, seen, see_foe
save

if (trace == '><') write(3,*) '>search'
search_on = .false.
DO i=1,is
!Find if searcher i detects any targets.
   see_foe = .false.
   IF (alive(i) .and. lambda(i,i) == INLOS) THEN
      ki = kind(i)
      lambda(i,i) = 1
```

```
      DO j=1,is
!     Find if searcher i detects target j.
         IF (i == j .or. .not.alive(j)) THEN! Ignore self and deead targets.
         ELSEIF (lambda(i,j) == INLOS) THEN
            rij = r(i,j)
            kj = kind(j)
!           Find appropriate column in table for exposure & motion.
               if (e(j,1) == HD) m = 5
               if (e(j,1) == TD) m = 5
               if (e(j,1) == FE) m = 6
               if (mision(j) == 'Attack') m = 7
            call hunt(tbl(1,1,kj,ki),7,rij,L)
            if (L < 1 .or. L > 7) call error ('Search: L must be 1..7. L=', L)
            y1 = tbl(L,m,kj,ki)
            y2 = tbl(L+1,m,kj,ki)
            pf = y1+(rij-tbl(L,1,kj,ki))*(y2-y1)/(tbl(L+1,1,kj,ki)-tbl(L,1,kj,ki))
            seen = ranu() < pf
            IF (seen) THEN! Tally & classify as friend or foe.
               lambda(i,j) = ISSEEN
               kappa(i,j,4) = kappa(i,j,4) + 1
               call history(i, j, ts, 'detect', 'search')
               call clasif(ts,i,j,ki,kj,see_foe)
            ELSE! Repeat search (unseen tgt is in range & in LOS).
               lambda(i,i) = INLOS
               search_on = .true.
            ENDIF
         ENDIF
      END DO
   ENDIF
   if (trace == 'search') write(3,*) i, (lambda(i,j), j=1,is)
   can_fire = (e(i,1) < FE) .or. (nbuddy(i) == 0)
   if (see_foe .and. can_fire) call select(ts,i)
END DO
if (search_on) call skedul(ts+1.0,0,'search',0, 'search')
if (trace == '><') write(3,*) '<search'
END
```

5.4 Clasif: Classify Target as Friend or Foe

Clasif finds whether the target is classified as a friend or as a foe. It first finds the range from i to j, then the appropriate column L in the table tbl. Next it interpolates in the table to find the probability of classifying the target as a foe. It then draws a random number to find the classification. This is recorded. Finally, if only a single battle is being played, **clasif** places one of four values in **str2**, depending on whether the target is a friend or foe and whether it was classified correctly. It then calls **history** to print the event if the target is a foe or a misclassified friend. It ignores correctly classified friends.

Arguments:

t_s	Simulation time (sec).
i, j	ID of searcher and target.
k_i, k_j	Kind of system i, j are.
see_foe	True if and only if i considers j a foe.

Locals:

foe	True if and only if i classifies j as a foe.
same	True if and only if i and j are on the same side.

r_{ij}	Range from i to j (km)
L	Row in table at which to interpolate.
p_f	Probability i classifies j as a foe.
n_s	Number of simulations (battles simulated)
$str2$	Character string containing these values:
	'enemy?' if friend incorrectly classified.
	'friend!' if friend is correctly classified.
	'enemy!' if foe correctly classified.
	'friend?' if foe is incorrectly classified.

```
SUBROUTINE CLASIF (ts,i,j,ki,kj,see_foe)
!Classify as friend or foe.
include 'global.h'
character str2*6
logical foe, same, see_foe

if (trace == '><') write(3,*) '>Clasif'
rij = r(i,j)
call hunt(tbl(1,1,kj,ki),7,rij,L)
pf = tbl(L,8,kj,ki) + (rij - tbl(L,1,kj,ki)) * &
   (tbl(L+1,8,kj,ki)-tbl(L,8,kj,ki)) /  &
   (tbl(L+1,1,kj,ki)-tbl(L,1,kj,ki))
foe = ranu() < pf
lambda(i,j) = ISFREN
if (.not.foe) kappa(i,j,6) = kappa(i,j,6) + 1
if (foe) kappa(i,j,7) = kappa(i,j,7) + 1
if (foe) lambda(i,j) = ISFOE
if (foe) see_foe = .true.
PRINT *,'Clasif: i,j,ki,kj=', i,j,ki,kj
IF (nn == nm) THEN
   same = narmy(i) == narmy(j)
   if (same .and. foe) str2='enemy?'
   if (same .and. .not.foe) str2='frend!'
   if (.not.same .and. foe) str2='enemy!'
   if (.not.same .and. .not.foe) str2='frend?'
   if(.not.same .or. foe) call history (i, j, ts, str2, 'clasif')
ENDIF
if (trace == '><') write(3,*) '<Clasif'
END
```

5.5 Pinpnt: Simulate Detection of Firing Signature

Pinpnt finds whether any system i saw system j fire. It skips any systems that don't have line of sight to j and any that don't pinpoint j. Pinpoint occurs only if a random draw is less than the probability that a system of kind i detects the firing signature of systems of kind j. If pinpoint occurs, that is recorded in λ_{ij} and scored in κ_{ij5} The event may be printed by **history. Clasif** classifies j as friend or foe of i. If i considers j a foe and can fire on it, **select** selects a target. System i can fire except if it is fully exposed and has a buddy in overwatch, in which case, it is bounding.

Arguments:

t_s	Simulation time (sec).

j	ID of system that fired.

Locals:

can_fire	False if and only if i is fully exposed and has a buddy. (The buddy provides overwatching fire while i bounds forward.)
i	ID of systems that may see j's firing signature.
k_i, k_j	Kind of system i and j are.

```
SUBROUTINE PINPNT (ts,j)
!Pinpnt: Find if any searchers detect j's firing signature.
include 'global.h'
logical can_fire, seefoe

if (trace == '><') write(3,*) '>pinpnt'
kj = kind(j)
DO i=1,is
   ki = kind(i)
   IF (i == j) THEN! Do nothing. System cannot pinpoint detect itself.
   ELSEIF (lambda(i,j) == INLOS .and. ranu() < atbl(15,ki)) THEN
      lambda(i,j) = ISSEEN
      kappa(i,j,5) = kappa(i,j,5) + 1
      call history(i, j, ts, 'pinpnt', 'pinpnt')
      call clasif (ts,i,j,ki,kj,seefoe)
      IF (lambda(i,j) == ISFOE) THEN
         can_fire = (e(i,1) < FE) .or. (nbuddy(i) == 0)
         if (can_fire) call select(ts,i)
      ENDIF
   ENDIF
END DO
if (trace == '><') write(3,*) '>pinpnt'
END
```

6. MOTION

6.1 Move: Update Position of Each System Each Second

Move is called at time zero to read the initial position and velocity of each system. On later calls, **detect_rg** updates the position of each combatant that is moving and finds the distance between each pair of combatants. The position is stored in the p array. Only live attackers are moving. **Move** does not update defenders or killed combatants

Move uses these variables and some global ones:

Arguments:

t	Time (sec). CHG
m	Read data if and only if $m = 1$.
n	Time between updates (hundredths of seconds).

Read in:

$mission_i$	Mission is defend or attack.
$name_i$	Kind of combatant (char).
o_{ik}	Initial position of ith system (km).
u_i	Maximum velocity of system (m/s).
v_{ik}	Velocity of ith system (m/s).

Local values:

i	ID of first system.
j	ID of second system.
i_o	local read error variable.
k	Coordinate index. 1=east, 2=north, 3=up.
$update_i$	True if and only if position of i needs to be updated. (At beginning of battle or if attacker is moving.)

Key global values:

n_s	# of combatants.
s_{ik}	kth coordinate of ith system.

If **detect_rg** is called and the second argument is $m = 1$, **move** reads data from the 'd.move' file and sets up initial values. It reads position and velocity data for each combatant and stores this information in o_{ik} (k=1..3) for the ith combatant. It also finds the distance from the ith to the jth combatant and places this value in r_{ij} as follows:

$$r_{ij} = \sqrt{(s_{i1}-s_{j1})^2 + (s_{i2}-s_{j2})^2 + (s_{i3}-s_{j3})^2}$$

The following is an example of the data **detect_rg** reads when it is first called.

```
DEPLOYMENT
SYS SIDE BUDDY X Y Z MISSION VMAX VX VY
M1      1 0 -0.1 -0.1 0. Defend 6.0 5.0 0.0
T80     2 3 3.1 -0.4 0. Attack -6.0 -5.0 0.0
T80     2 2 3.1 -0.2 0. Attack -6.0 -5.0 0.0
END
F/Run/d.move
```

Whenever **detect_rg** is called, it updates positions linearly. The dimensions of s and o are kilometers and v_c is meters per second, so the latter is multiplied by 0.001 to convert to km/s. **Move** uses the following equation to move combatants on a straight line path:

$$s_{i1} = o_{i1} + 0.001 v_{i1} t$$

```
SUBROUTINE MOVE (ts,m)
! Find where systems are.
include 'global.h'
character namei(IX)*6
logical update(IX)
real o(IX,3)
save
1 format(a,i5)
2 format(10a8)
3 format (a,i5,10f5.3)
f(i,j) = sqrt((s(i,1)-s(j,1))**2 + (s(i,2)-s(j,2))**2 + (s(i,3)-s(j,3))**2)

if (trace == '><') write(3,*) '>move'
IF (m == 0) THEN! Read deployment data.
    open (2, file='d.move', status='old')
    rewind 2
    read(2,*)
    read(2,*)
    i = 0
    DO! Read combatant data.
      i = i+1
      read(2,*,iostat=io) namei(i), narmy(i), nbuddy(i), &
          (o(i,j), j=1,3), mision(i), u(i), v(i,1), v(i,2)
    IF (io /= 0 .or. namei(i) == 'END') EXIT
    END DO
    close (2)
! Set initial values.
    is = i - 1
    DO i = 1, is
      r(i,i) = 0.0
      kind(i) = ialf(name,ks,namei(i))
      if (kind(i) == 0) call error ('MOVE: no data for:'//namei(i), i)
      moving(i) = mision(i) == 'Attack'
    END DO
    write(3,1) '# of actual combatants', is
    write(3,2) (namei(i), i=1,is)
    write(3,'(10i8)') (narmy(i), i=1,is)
    write(3,'(10f8.2)') (o(i,1), i=1,is)
    write(3,'(10f8.2)') (o(i,2), i=1,is)
    write(3,'(10f8.2)') (o(i,3), i=1,is)

ELSE! Find where all systems are.
    DO i = 1, is
      update(i) = (mision(i) == 'Attack' .and. moving(i)) .or. m ==1
      if (ts == 0) g(i) = 0.0
```

```
      if (ts == 0) h(i) = 0.0
      if (update(i)) s(i,1) = o(i,1) + 0.001*v(i,1)*(ts-g(i))
      if (update(i)) s(i,2) = o(i,2) + 0.001*v(i,2)*(ts-g(i))
! Find range from system i to system j.
    DO j = 1, i-1
      if (update(i) .or. update(j)) r(i,j) = f(i,j)
      r(j,i) = r(i,j)
    END DO
    if (trace == 'move') write(3,3) 'rg from', i, (r(i,j), j=1,i)
    END DO
    call skedul (ts+5.0,0, 'move  ', 0, 'move  ')
ENDIF
if (trace == '><') write(3,*) '<move'
END
```

6.2 Bound: Have Tank Bound Forward or Start Overwatch

U.S. Army tankers are taught to advance using bounding overwatch tactics. The tanks are divided into pairs and one of them bounds forward while the other provides covering fire from a hull defilade overwatch position. When Force Wars reads the **d.move** file, it reads the pairings in $nbuddy_i$. $Nbuddy_i$ must be set to j, where $nbuddy_j = i$.

At the beginning of each battle (replication), one of the pair is placed in hull defilade overwatch. It can acquire and fire on targets. The other bounds forward. It is fully exposed and does not acquire targets. After the bounding tank covers a predetermined distance, it goes into overwatch and its buddy bounds forward.

This tactic may be applicable to other vehicles or other tactics may need to be designed for them. It is not clear what a tank in bounding overwatch will do when its buddy is killed. In Force Wars, a tank continues to alternately bound forward and assume an overwatch position even after its buddy dies.

Variables are:

Arguments:

i	Combatant
n	= 0 to start battle, = 1 thereafter.
t_s	Simulation time (sec).

Locals:

j	ID of firer's buddy.

Key globals:

$owatch_i$	True if and only if i is in overwatch.
e_{i3}	Exposure of i due to bound / overwatch.
h_i	Time bounding systems halts to begin overwatch (sec).
g_i	Cumulative delay time along path (sec).
$moving_i$	True if and only if the system is moving.
$iseg_i$	Index of in-view or out-of-view segment lengths. Used to find if combatant is in view or out of view.

The value *nbuddy*$_i$ is the key input controlling bounding overwatch. It contains a zero if the ith system is not using bounding overwatch and the ID of the buddy if the pair use bounding overwatch. When each replication begins, one of the pair is randomly selected to be in overwatch by **init2**.

```
SUBROUTINE BOUND (i,n,ts)
include 'global.h'
common /terane/ del(100), a(IX,3), q(IX), iseg(IX)

IF (n == 0) THEN! Initialize bounding
   owatch(i) = ranu() < 0.5
   j = nbuddy(i)
   owatch(j) = .not. owatch(i)
   call skedul (ts, i, 'bound ', NULL, 'bound ')
   call skedul (ts, j, 'bound ', NULL, 'bound ')
ELSEIF (alive(i)) THEN! Flip-flop tank between bounding and overwatch
   owatch(i) = .not.owatch(i)
   IF (owatch(i)) THEN! Have system i settle into overwatch.
      call history (i, 0, ts, 'owatch', 'bound ')
      e(i,3) = HD
!     Halt i.
         h(i) = ts ! Halt time (sec).
         moving(i) = .false.
      if (any(lambda(i,:)==ISFOE)) call select(ts,i)
!     Cancel appear or vanish as appropriate.
         if (mod(iseg(i),2)==0) call cancel(i,'appear',0,'bound ')
         if (mod(iseg(i),2)==1) call cancel(i,'vanish',0,'bound ')
   ELSE! Have system i bound foreward.
      call history (i, 0, ts, 'bound ', 'bound ')
      e(i,3) = FE
!     Move i.
         g(i) = g(i) + ts - h(i)! Cumulative delay time (sec).
         moving(i) = .true.
!     Schedule appear or vanish as appropriate.
         if (mod(iseg(i),2)==0) call skedul(ts,i,'appear', NULL, 'bound ')
         if (mod(iseg(i),2)==1) call skedul(ts,i,'vanish', NULL, 'bound ')
   ENDIF
   e(i,1) = min(e(i,2),e(i,3),e(i,4))! Overall exposure.
   call skedul (ts+30.,i,'bound ', NULL, 'bound ')
ENDIF
END
```

7. TARGET EXPOSURE

This section discusses the exposure of the target and the routines that simulate it. The model considers a number of factors. If the exposure changes, the probability of detection changes. If the target vanishes, missiles will be aborted and defending firers may pop down or switch to another target. If a defending firer is using pop-up/pop-down tactics, he may pop down after firing two rounds, displace sideways and pop back up to turret defilade. When it engages a target, it may go from turret defilade to hull defilade.

Attackers may use bounding overwatch tactics, in which case, a pair of combatants will alternate between advancing (moving and fully exposed) and overwatch (stationary and hull defilade). Attackers will also alternate between being masked and unmasked by terrain. The model does not currently simulate smoke but will likely be extend in the future to do so.

The routines that simulate these conditions are:

pop_dn	Go from hull defilade to full defilade and then to turret defilade.
pop_up	Go from turret defilade to hull defilade.
bound	Go from stationary, hull defilade to moving, fully exposed.
terain	Initialize terrain in-view/out-of-view segment lengths at battle start.
vanish	Find if moving combatant goes behind terrain.
vanter	Simulate effects of vanishing behind terrain.
appear	Find if moving combatant comes from behind terrain.
aprter	Simulate effects of appearing from behind terrain.

Exposure values are stored in a table e, where e_{im} contains one of the following values:

FD = 1	Full defilade
TD = 2	Turret defilade
HD = 3	Hull defilade
FE = 4	Fully exposed

The first subscript is the system number. The second indicates:

m = 1 Overall exposure
m = 2 Pop-up/pop-down status of defender i
m = 3 Bounding overwatch status of attacker i
m = 4 Terrain mask status of attacker i

At the beginning of each battle, **init2** sets e to fully exposed (FE) then for each defender it resets $e_{i1} = e_{i2} = TD$ Finally, **init2** calls **bound** for further initialization of array e for the attackers.

At this time, **Bound** randomly sets one of each pair in bounding status and the other in overwatch. The exposure of the bounding system is then set to $e_{i1} = e_{i3} = FE$ and the exposure of the overwatch system is set to $e_{i1} = e_{i3} = HD$.

e_{i3} tells whether entity i is FE or FD due to terrain. *Note: los checks if* $e_{i1} > FD$ *and* $e_{i2} > FD$. Routine **bound** sets $e_{i3} = HD$ when an entity settles into overwatch and sets $e_{i3} = FE$ when the entity begins to bound forward. Routine **pop_dn** sets $e_{i1} = e_{i2} = FD$ when the entity pops down, and then sets it back to $e_{i2} = TD$ after disengaging those that had targeted entity i.
Similarly, routine **pop_up** sets $e_{i:} = HD$ when entity i pops up.

Initially, defenders are popped up in turret defilade ($e_{i2} = TD$). They are paired. One of the pair selects a target and goes to hull defilade. It fires perhaps two rounds at the target and pops down to full defilade, offsets laterally to another prepared position and pops back up to turret defilade.

If the defender does not use this tactic, it will remain in hull defilade at all times.

The attacker may bull through, in which case, it does not stop and assume an overwatch position, so it is always fully exposed (e_{i3}=FE)

7.1 Pop Up and Pop Down.

The code simulates a defender that pops down after firing a fixed number of rounds, moves laterally, and pops back up. This is for defenders firing bullets and other fire-and-forget rounds. It really models the time required to do this and breaking of lines of sight. The line-of-sight breakage means loss of acquisitions and disengagements. The code doesn't actually relocate the tank laterally. That's not important.

Tankers are trained to pop down after two shots, but the actual number is a local command decision. A tank platoon of four tanks is divided into two pairs. When a tank is detected, one of the pair moves from TD to HD and fires two shots at the target while the partner observes from TD. The firer moves back into full defilade when the target is killed or immediately after firing the second round, whichever occurs first. It then moves laterally and moves up to a TD position and observes for its buddy. Meanwhile, the buddy has moved to HD, fired one or two rounds and gone to full defilade. The process continues to cycle like this.

This code will be valid for tanks, Infantry Fighting Vechicles (ITVs), and ITVs. It may be valid for helicopters except that I don't plan to have them buddy up. It is probably not valid for M113s and dismounted infantry.

The code uses the following variables:

Arguments:
t_s Simulation time (sec).

i ID of weapon system that may pop down.

Locals:

dt Time delay for lateral motion and pop up to TD (sec).
e_i Exposure of tank i (=FD, TD, HD, FE)
nrf_i Number of rounds fired at current target (reset).
j Combatants that might be observing i.

Here is the code for pop up:

```
SUBROUTINE POPUP (ts,i,j)
!Popup: Simulate tank or chopper pop-up  and schedule consequences. schedule fire on j.
include 'global.h'

if (trace == '><') print *, '>popup'
e(i,:) = HD
dt = 10.8*exp(rolln(0.5))
call skedul(ts+dt,i,'fire  ',j, 'popup ')
if (trace == '<<') print *, '>popup'
END
```

Here's the code to simulate pop down:

```
SUBROUTINE POPDN (ts,i)
!Popdn: Simulate tank or chopper pop-down and shedule con == ences.
include 'global.h'

if (trace == '><') print *, '>popdn'
IF (e(i,2) < FE) THEN! Have all but fully exposed tanks pop down & back to TD.
! Have system pop down.
    e(i,1:2) = FD
    NRF(i) = 0
    call diseng(ts,1,is,i,INSIDE)
! Have system pop back up to turret defilade.
    e(i,1:2) = TD
! Have all observers lose system i and regain line-of-sight.
    DO j=1,is
      IF (e(j,1) /= FD) THEN
        if (lambda(i,j) == INSIDE) lambda(i,j) = INLOS
        if (lambda(i,j) == INSIDE) lambda(i,i) = INLOS
        if (lambda(j,i) == INSIDE) lambda(j,i) = INLOS
      ENDIF
    END DO
! Have system i resume search (& others too).
    lambda(i,i) = INLOS
    if (.not.search_on) call skedul(ts,0,'search',0, 'popdn ')
    search_on = .true.
ENDIF
if (trace == '><') print *, '<popdn'
END
```

7.2 Terain: Find Segments Where Attacker is Masked by Terrain.

Terain finds the portions of the attacker paths where the attackers are hidden from the defenders by terrain.

Init calls this routine at the beginning of each battle. **Terain** then creates a table ∇_{100} and puts randomly chosen in-view segment lengths in the odd elements of ∇ and out-of-view segment lengths in the even elements of ∇ as shown in Figure 1.

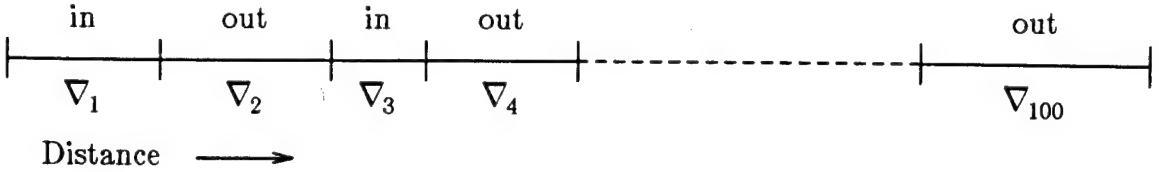


Figure 1. Alternating in view and out of view segments.

Later, the code will need to know which segment each attacker is in, the length of the segment, and where the segment began. The final portion of the routine stores this information and tentatively schedules a **vanish** for each moving combatant. (The **vanish** is only tentative because the attacker may stop while traversing the in-view segment, it may halt to fire or it may be mobility killed.)

The segment lengths are random variates drawn from Wiebull distributions. The in-view segment length is:

$$f_2 = \alpha_1 f^{\beta_1}$$

and the out-of-view segment length is:

$$f_2 = \alpha_2 f^{\beta_2},$$

where

$f = -\log(ran)$, and
ran is a draw from the standard uniform distribution.

Sometimes these segment lengths are excessively long so that the attackers are out of view at all reasonable engagement ranges, with the result that no engagement occurs. For this reason, the segment lengths are truncated to 1 km.

Locals:

α_1	Coefficient of Wiebull equation for in-view segment lengths (km).
α_2	Coefficient of Wiebull equation for out-of-view segment lengths (km).
β_1	Exponent of Wiebull equation for in-view segment lengths.
β_2	Exponent of Wiebull equation for out-of-view segment lengths.
f_2	Temporary random length (km).
∇_j	Length of jth segment (km).

Key globals:

a	Position of combatant at beginning of current segment (km).
s	Current position of combatant (km).
q	Length of segment that the combatant is currently on (km).
n_s	Number of current battle (replication).
∇	Vector of segment lengths (m).
$iseg_j$	Segment combatant j is currently on.

SUBROUTINE TERAIN

```
! Terain: draw alternating in-view, out-of-view segment lengths.
! Local variables:
implicit none
include 'global.h'
integer i ! # of combatant.
integer j ! # of intervisibility segment.
real f    ! temporary variable
real f2       ! segment length (km).
! Global variables specific to terain, appear, aprter, vanish, vanter:
real a1, b1, a2, b2
integer iseg
real a    ! Position when ith combatant vanished or appeared (km).
       ! a(1,i) East
       ! a(2,i) North
       ! a(3,i) Height
real q ! q(i) Distance to travel before vanishing or appearing.
real del  ! del(j) jth segment length (km).
REAL RG0, ranu
common /terane/ del(100), a(IX,3), q(IX), iseg(IX)
common /terrac/ a1, b1, a2, b2

if (trace == '><') write(3,*) '>terain'
a1 = 0.3
a2 = 0.75
b1 = 1.
b2 = 2.

DO j=1,99,2! Randomly select 50 pairs of in-view, out-of-view segment lengths (km)
   f = -alog(ranu())
   f2 = a1*f**(1./b1)
   del(j) = max(0.002, min(f2,1.0))
   IF (del(j) <0.001) PRINT *,'Terain: j,f,del(j)=', j, f, del(j)
   f = -alog(ranu())
   f2 = a2*f**(1./b2)
   del(j+1) = max(0.002, min(f2,.3*rg0))
END DO

IF (nn == nm) THEN
   write(3,*) 'In view and out of view segment lengths (m):'
   write(3,*) ' In Out  In Out  In Out  In Out  In Out  In Out  In Out  In Out  In Out  In Out
   write(3,'(20i4)') int(1000*del)
END IF

! Save position when system vanished and distance to travel before appearing.
   a = s
   q = del(1)
   iseg = 1
DO i=1,is ! Schedule each to vanish.  ----------------------------------WRONG!
   if (moving(i)) call skedul (0.,i,'vanish',NULL, 'terain')
END DO
if (trace == '><') write(3,*) '<terain'
END
```

7.3 Appear: Simulate or Reschedule an Appear Event.

Appear simulates re establishing line of sight between a target and one or more searchers.

The overall structure of the routine is as follows:

simulate appearance from behind terrain
  find distance tank has traveled
  IF (tank has traversed entire out-of-view distance) THEN

treat appearance of the tank
ELSE
reschedule appearance
ENDIF

The routine finds how far the tank has traveled since it vanished. If the tank has traversed the entire out-of-view segment length, it's ready to appear, otherwise **appear** will estimate the remaining time to finish the out-of-view segment and reschedule itself for that time.

Arguments:

t_s	Simulation time (sec).
j	Combatant.

Locals:

travel	Distance traveled in this segment (km).
dt	Estimated time to complete segment.

Key globals:

s_j	Current position of j (km).
a_j	Position at beginning of intervisibility segment (km)
$iseg_j$	Index of segment j is in.
∇_n	Length of nth segment (km).
q_j	Length of segment j is in (km).
$alive_j$	True if and only if j is alive.
e_{j3}	Exposure due to bounding overwatch.

```
SUBROUTINE APPEAR(ts,j)
! If tgt appears treat, otherwise reschedule appearance
implicit none
include 'global.h'
integer j, iseg
real ts
real del, a, q, travel, x, y, dt, rss
common /terane/ del(100), a(IX,3), q(IX), iseg(IX)
rss(x,y) = sqrt(x*x+y*y)
1 format (a,f8.2,i2,3f8.3,f8.2)

if (trace == '><') write(3,*) '>appear'
travel = rss(s(j,1)-a(j,1), s(j,2)-a(j,2))
q(j) = del(iseg(j))
IF (travel > q(j)) THEN ! Tgt is no longer masked by terrain
   call history (j, 0, ts, 'does  ', 'appear')
   a(j,:) = s(j,:)
   iseg(j) = iseg(j)+1
   if (iseg(j) > 100) iseg(j)=iseg(j)-100
   q(j) = del(iseg(j))
   call aprter(ts,j,FE)
!  Schedule next disappearance
      dt = 1000.0*q(j)/abs(u(j))
           if (dt<1.0) dt = 1.0
```

```
      call skedul(ts+dt,j,'vanish',NULL, 'appear')
ELSE ! Reschedule appearance. Tgt hasn't reached end of mask yet.
   call history (j, 0, ts, 'doesnt', 'appear')
   IF (alive(j) .and. e(j,3) == FE) THEN
      dt = 1000.0*(q(j) - travel) / abs(u(j))
      if (dt<1.0) dt=1.0
      call skedul (ts+dt,j,'appear',NULL, 'appear')
   ENDIF
ENDIF
if (trace == '><') write(3,*) '<appear'
IF (del(1) <0.001) THEN
   print*,'Appear: del(1), j, iseg(j)=', del(1), j, iseg(j)
ENDIF
END
```

7.4 Aprter: Simulate Target Appearing from Behind Terrain.

The combatant has just re-appeared from behind terrain. **Aprter** sets the attacker to fully exposed and re-establishes all lines of sight.

Arguments:

t_s	Simulation time (sec).
j	Combatant.
jexpos	Exposure of combatant.

Locals:

i	Other combatant.

Key globals:

e_{j4}	Exposure of j due to terrain.
e_{j1}	Overall exposure.

```
SUBROUTINE APRTER(ts,j,jexpos)
!Aprter: Tgt has appeared from behind terrain, reset.
implicit none
include 'global.h'
integer j, i, jexpos
real ts

if (trace == '><') write(3,*) '>aprter'
e(j,4) = jexpos
e(j,1) = min(e(j,2),e(j,3),e(j,4))
! Restore all lines-of-sight involving j
  DO i=1,is
    IF (e(i,4) > FD .and. i /= j) THEN
      call los (ts, i, j)
      call los (ts, j, i)
    ENDIF
  END DO
if (trace == '><') write(3,*) '<aprter'
END
```

7.5 Vanish: Simulate or Reschedule Vanish Event.

Vanish models the disappearance of a target due to terrain blocking the line-of-sight. Terrain blocks the line-of-sight only when the attacker traverses the in-view segment, so **vanish** is only scheduled tentatively for terrain blockage. The code checks to

see if the attacker has completed the in-view segment. If so, it schedules a subsequent **appear**, otherwise it reschedules **vanish** based on the in-view distance left to travel and the combat cruise speed of the attackers.

If the attacking tank has completed the in-view segment, the code sets up the next **appear** event and calls **vanter** to complete the **vanish** event. To set up the next **appear** event, the code records the beginning of the out-of-view segment, the segment number, and the length of the segment. It then finds the time to complete the out-of-view segment and schedules an **appear** event at the end of that time.

Arguments:

t_s	Simulation time (sec).
j	Combatant.

Locals:

travel	Distance traveled in current segment (km).
q_j	?
dt	Time to reach end of current segment (sec).

Key globals:

a_j	Position of j at beginning of current segment (km).
s_j	Current position of j (km).
$iseg_i$	Segment j is in.
∇_n	Length of segment n (km).
$alive_j$	True if and only if j is alive.
e_{j3}	Exposure of j due to bounding overwatch.
n_n	Battle number.
n_m	Battle number to monitor.

```
SUBROUTINE VANISH(ts,j)
! V8.2If tgt vanishes treat, otherwise reschedule vanish
implicit none
include 'global.h'
integer j, iseg
real del, a, q, rss, travel, x, y, ts, dt
common /terane/ del(100), a(IX,3), q(IX), iseg(IX)
rss(x,y)=sqrt(x*x+y*y)
1 format (a,f8.2,2i3,2f6.3,f6.1)

if (trace == '><') write(3,*) '>vanish', '  ts,j=', ts,j
!Terrain causes intervisibility
travel = rss(s(j,1)-a(j,1), s(j,2)-a(j,2))
q(j) = del(iseg(j))
IF (trace == 'vanish') THEN
   print 1, 'Vanish: ts,j,seg,q,travel,u=',  ts, j, iseg(j), q(j), travel, u(j)
ENDIF
IF (travel > q(j)) THEN ! Tgt is now masked by terrain
   if (nn == nm) call history (j,0,ts,'does  ','vanish')
   a(j,:) = s(j,:)
   iseg(j) = iseg(j)+1
   if (iseg(j) > 100) iseg(j)=iseg(j)-100
   q(j) = del(iseg(j))
```

```
    call vanter(ts,j)
    dt = 1000.*q(j)/abs(u(j))
    if (dt < 1.0) dt=1.0
    IF (ts+dt>2000.) WRITE(3,*) 'Vanish1: j,ts,dt,q(j),u(j)=', j,ts,dt,q(j),u(j)
    call skedul (ts+dt,j,'appear',NULL, 'vanish')
ELSE IF (alive(j) .and. e(j,3) == FE) THEN ! Not yet masked by terrain, so reschedule
    if (nn == nm) call history (j,0,ts,'doesnt','vanish')
    dt = 1000.0*(q(j) - travel) / abs(u(j))
    if (dt<1.0) dt = 1.0
    IF (ts+dt>2000.) WRITE(3,*) 'Vanish2: j,ts,dt,q(j),u(j)=', j,ts,dt,q(j),u(j)
    call skedul (ts+dt,j,'vanish',NULL, 'vanish')
ELSE
    if (nn == nm) call history (j,0,ts,'doesnt','vanish')
ENDIF
if (trace == '><') write(3,*) '<vanish: ts,j,moving(j)=', ts,j,moving(j)
END
```

7.6 Vanter: Treat Target Vanishing Behind Terrain.

The target has now definitely vanished behind terrain. It is marked as being in full defilade, as having no targets, and no detections. All lines of sight to and from it are broken. The target sees no foes and they no longer see it. The last fire times are reset to zero so the target and its foes are treated as new threats when the target reappears. Any missiles are aborted. Foes engaging the target disengage it.

Arguments:

t_s	Simulation time (sec).
j	Combatant.

Key globals:

e_{j4}	Exposure of j due to terrain.
e_{j1}	Overall exposure of j.
i_s	Number of combatants.
$nchan_j$	Number of busy guidance channels.

```
SUBROUTINE VANTER (ts, j)
! Have target j vanish behind terrain.
implicit none
integer j
real ts
include 'global.h'

if (trace == '><') write(3,*) '>vanter'
call history (j, 0, ts, 'vanish', 'vanter')
e(j,4) = FD
e(j,1) = FD
! ndet(j) = 0
! busy(j) = .false.
!Cancel all lines-of-sight and sightings from firer j to all targets.
   call diseng (ts, 1, is, j, INSIDE)
!  DO i=1,is
!     tfire(j,i) = 0.0
!     tfire(i,j) = 0.0
!  END DO
! Abort outgoing missiles
!* call abort(t,j,ALLx)
   nchan(j) = 0
! Abort incoming rounds & disengage firers with LOS to target j.
```

```
   call cancel (j,'fire  ',NULL,'vanter')
   call cancel (j,'select',NULL,'vanter')
if (trace == '><') write(3,*) '<vanter'
END
```

7.7 LOS: Line of Sight.

Los finds if line-of-sight exists. First, it checks λ_{ij} to see if j is inside the detection range of i. Second, it checks $e_{i,1}$ and $e_{j,1}$ to find if i and j are at least partly exposed (not in full defilade). If all these conditions are satisfied, **los** establishes line-of-sight from i to j. If this is just done, λ_{ii} is set so that i can begin searching. If search is not on, it is turned on (scheduled).

The code uses the following variables:

Arguments:

t_s	Simulation time (sec).
i	Firer
j	Target

Locals:

L_o	Prior value of λ_{ij}.

Key globals:

λ_{ij}	Connection between firer i and target j.
κ_{ij3}	Count of times line-of-sight is established.
search_on	True if and only if search is on. Inhibits multiple scheduling of search.

```
SUBROUTINE LOS (ts,i,j)
!See if line-of-sight exists between i and j.
include 'global.h'
integer Lo

if (trace == '><') write(3,*) '>los, ts,i,j=',ts,i,j
Lo = lambda(i,j)
if (lambda(i,j) == INSIDE .and. e(i,1) > FD .and. e(j,1) > FD) lambda(i,j) = INLOS
IF (lambda(i,j) == INLOS) THEN
  lambda(i,i) = INLOS
  kappa(i,j,3) = kappa(i,j,3) + 1
  call history (i, j, ts, 'has   ', 'los   ')
  IF (.not.search_on) THEN
    call skedul(ts,0,'search',0, 'los   ')
    search_on = .true.
  ENDIF
ELSE
  if (Lo > INSIDE) call history (i, j, ts, 'lost  ', 'los   ')
ENDIF
if (trace == '><') write(3,*) '<los, lambda(i,j), search_on=',lambda(i,j), search_on
END
```

8. WEAPON/AMMUNITION AND TARGET SELECTION

The model was constructed using these assumptions:

1. There may be as many as 3 weapon systems on a platform.
2. Only one weapon can be used at a time.
3. Any of the three weapons can be a fire and forget system (guns and some missiles).
4. Only the first weapon can be a guide to impact system (most missiles).
5. If the missile must be guided to impact, the missile system may be able to fire at multiple targets simultaneously using multiple guidance channels.

There are several ways to select ammunition/weapons and targets. The possibilities include: 1) Select the most lethal ammunition and the most vulnerable target, 2) elect the most lethal ammunition and the most lethal target, 3) use some combination of these, 4) use the selection priorities that the gunners are trained to use.

The model currently uses the first method. The second and fourth methods may be quite similar and more realistic, and the third could be optimal. None of these consider communications between tanks to spread fire nor preplanned methods of spreading fire evenly over the targets.

If the most lethal target were selected, the following might be considered: 1) choose closer rather than more distant targets, 2) choose halting to fire systems rather than those that continue to move, 3) choose those that are pointing at you instead of those that are not, 4) choose one that has recently fired because it is more likely to be alive, and 5) choose one that has weapons that can destroy you over one that does not.

8.1 Select: Find What Should be Selected and Schedule Firing

Select has two parts. First, it attempts to select a weapon or ammunition type and a target. If no weapon has been selected, **select** calls **selec2** to select both a weapon (ammunition) and a target. If weapon 1 has been selected, it calls **selec2** to select a target. If this is successful, it schedules systems in turret defilade move up to hull defilade and schedules firers in hull defilade or fully exposed to fire.

Arguments:

i	ID of searcher.
t_s	Simulation time (sec).

Locals:

j	ID of target selected.
$k = kind(i)$	Kind of system.
t_f	Time to fire round (sec).

The code is:

```
SUBROUTINE SELECT (ts,i)
!Select: Pick the weapon and target.
include 'global.h'

if (trace == '><') write(3,*) '>select. ts,i=', ts, i
k = kind(i)
IF (nwpn(i) == 0) THEN! Pick from any weapon and target.
   call selec2(i,j,3,nwpn(i))
ELSEIF (nwpn(i) == 1 .and. nchan(i) < itbl(7,k)) THEN
!Pick a tgt but use guided missile system. GC is available.
   call selec2(i,j,1,nwpn(i))
ELSE! Do not select.
   j = 0
   call history(i, j, ts, '(busy) ', 'select')
ENDIF

IF (j == 0) THEN
ELSE! Schedule popup or firing.
   call history(i, j, ts, 'select','select')
   kappa(i,j,8) = kappa(i,j,8) + 1
   tf = atbl(14,k)*exp(rolln(0.5))
   if (e(i,2) == TD) call skedul (ts+2.0,i,'popup ',j, 'select')
   if (e(i,2) >= HD) call skedul (ts+tf,i,'fire  ',j, 'select')
   if (nwpn(i) == 1 .and. itbl(7,k) > 0) nchan(i) = nchan(i) + 1
ENDIF
if (trace == '><') write(3,*) '<select'
END
```

8.2 Selec2: Select Weapon/Ammunition and Target

Selec2 does the actual selection. It finds the probability of killing each target with each type of ammunition and chooses the target and ammunition that gives the highest kill probability.

Arguments:

i, j_o	ID of firer and selected target.
n_w	# of weapons to choose from.
k_o	ID of selected weapon.

Locals:

q_o	Highest kill probability so far.
j	ID of target under consideration.
l	Interpolation index.
k	# of weapon/ammunition under consideration.
q	Kill probability of ammunition k against target j
y_1, y_2	Kill probabilities at ranges l, l+1.

```
SUBROUTINE SELEC2 (i,jo,nw,ko)
!Discard targets for which there is no good ammo.
!i, j, k, l are firer, tgt, ammo, and range indices.
include 'global.h'
real E_(7)
```

```
DATA E_ /0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0/

if (trace == '><') write(3,*) '>selec2'
qo = 0.0
jo = 0
DO j=1,is
   IF (i /=j .and. lambda(i,j) == ISFOE) THEN! Find probability of killing target j.
      rij = r(i,j)
      call hunt (e_,nr,rij,l)
      DO k=1,nw ! Find probability of killing target j with ammo k.
         IF (nrd(i,k) > 0) THEN! Ammo type k is available.
            IF (L == 7) THEN
               q = p(7,k,kind(j),kind(i))
            ELSE
               y1 = p(l,k,kind(j),kind(i))
               y2 = p(l+1,k,kind(j),kind(i))
               q = y1 + (r(i,j)-E_(l)) * (y2-y1) / (E_(l+1)-E_(l))
            END IF
            IF (q > qo) THEN! Replace old choice. Firer i does better vs tgt j w/ ammo k.
               qo = q
               jo = j
               ko = k
            ENDIF
         ENDIF
      END DO
   ENDIF
END DO
if (jo > 0) lambda(i,jo) = ISTGT
if (trace == '><') write(3,*) '<selec2'
END
```

9. FIRE ROUTINES

The routines that simulate the firing of a round are **fire, fired_single, fired_missile,** and **fired_burst.**

Fire decrements ammunition, calls **pinpoint** to find if the muzzle flash is detected, schedules **impact** of the round, and calls the appropriate one of the other fire routines. **Fired_single** completes the simulation of firing a single, fire-and-forget round such as a bullet. It schedules firing of the next shot if ammunition is available and selects a new round type and target otherwise. If all ammunition is used, it schedules a **hide** event. **Fired_burst** is a dummy routine which can be expanded to simulate burst firing weapons. **Fired_missile** completes the simulation of firing a guided missile. If all missiles have been used, it is time to pop down, or to switch targets, this event routine calls **diseng** to disengage the firer from the target. Otherwise, it schedules the next round to be fired.

9.1 Fire: Simulate Firing of a Bullet or Missile.

Fire decrements ammunition, finds who detects the firer's muzzle flash, records the times, schedules the impact of the round, and calls the appropriate subsidiary routine to find what happens next.

To find what happens next requires consideration of the following questions:

1. Is there more of the currently selected ammunition on board?
2. If so, what is the firing policy? Continue firing until the target is killed? Continue firing until a fixed number of rounds are shot at the target? Continue firing until the firer hits the target?
3. Given that the firer is to shoot again at the current target, when will the next round be launched? This depends on whether the weapon is a missile or gun.
4. If it is a gun, does it fire single shots or bursts?
5. If it fires single shots, is the loader a manual loader, a load assist device, or an automatic loader?
6. If it is a missile system, does it have a single guidance channel or does it have multiple guidance channels so that it can guide n missiles to n targets simultaneously?
7. If it is a missile with multiple guidance channels, are all the guidance channels busy?
8. If there are no more rounds of the currently selected type on board, are there other rounds on board?
9. Is there a target for which these other rounds are lethal?
10. If the firer still has rounds but there are no targets or targets the firer has no ammunition to defeat, does the firer proceed with the force or seek cover?

Arguments:

t_s	Simulation time (sec)
i	Firer
j	Target
k	Kind of weapon/ammunition
k_i	Kind of firer eg ki=1 could be M1A1

v_m	Muzzle velocity (m/s).
β	Siacci coefficient (hz).
n	Firer with sign (<0 for fire & forget. >0 for guided missiles).
n_c	# of guidance channels (when using guided missiles).
$t_f = r_{ij}/(v_m - \beta * r_{ij})$	Siacci equation Time of flight (s).

```
SUBROUTINE FIRE (ts,i,j)
! Fire: Simulate firing a round.
implicit none
real ts
real beta, vm, tf
include 'global.h'
integer i, j, k, ki, n, nc

if (trace ==  '><') write(3,*) '>fire'
!Update round counts, time of last fire.
   k = nwpn(i)          ! Weapon i is using.
   nrd(i,k) = nrd(i,k) - 1! Rounds remaining of kind k for i.
   nrf(i) = nrf(i) + 1    ! rounds i has fired at the current target.
   kappa(i,j,9) = kappa(i,j,9) + 1! Count of rounds i fired at j.
   call pinpnt(ts,i)      ! Find who sees muzzle flash and smoke.
   ki = kind(i)
   vm = atbl(k+7,ki)      ! Muzzle velocity of round k for i.
   beta = atbl(k+10,ki)   ! Siacci coefficient of round k for i.
   tf = r(i,j)/(vm-beta*r(i,j))! Time of flight to range r.
if (trace ==  'fire') write(3,*) 'Fire: vm, beta, tf=',vm,beta,tf
nc = itbl(7,ki)           ! # of guidance channels.
n = i               ! Positive sign allows cancelling guided missiles.
if (k>1 .or. nc==0) n = -i! Negative sign avoids cancelling fire & forget rounds.
call skedul (ts+tf, n, 'impact', j, 'fire  ')
if (trace ==  'fire') write(3,*)i, ' has', nrd(i,k), ' more rds of type', k
IF (k == 1 .and. nc > 0) THEN! Using wpn 1 with guidance channels.
   if (nchan(i) < nc) call fired_missile (ts,i,j)
ELSE
   call fired_single (ts,i,j,ki)
ENDIF
!  call fired_burst (ts,i,j,ki)
if (trace ==  '><') write(3,*) '<fire'
END
```

9.2 Fired_single: Schedule Results of Firing a Round

Fired_single schedules events that occur after a round is fired. It either disengages the target or schedules firing of another round at the target. It disengages if it has no more of the kind of rounds it just fired, if pop- down criteria have been satisfied, or target switching criteria have been satisfied.

Arguments:

t_s	Simulation time (s).
i	Firer
j	Target

Locals:

n_f	# rounds fired at current target.

n_p	# rounds to fire before popping down.
n_s	# rounds to fire before switching targets.
n_a	# rounds available of the kind just fired.
t_v	# variable times associated with loading.
t_c	# constant time associated with loading.

```
SUBROUTINE FIRED_SINGLE (ts,i,j,k)
! Fired_single: Move, fire, or switch targets after firing single shot.
implicit none
real tc, tv, rolln, ts
integer i, j, k, nf, np, na, nd
include 'global.h'

if (trace == '><') write(3,*) '>fire_s'
nf = nrf(i)                ! # rounds fired at current target.
np = itbl(17,k)            ! # rounds to fire before popping down.
nd = itbl(18,k)            ! # rounds to fire before switching.
na = nrd(i,nwpn(i))        ! # rounds available of selected type.
IF (na == 0 .or. nf == np .or. nf == nd) THEN
   call diseng (ts,i,i,j,ISFOE)
ELSE                  ! Schedule next shot.
   tv = atbl(6,k)
   tv = tv*exp(rolln(0.5))! Manual loader or other variable time.
   tc = 0.0            ! MUST READ IN A VALUE LATER!
   if (itbl(16,k) == 2) tv = tv + tc! Load assist device.
   if (itbl(16,k) == 3) tv = max(tv, tc)! Automatic loader.
   call skedul (ts+tv,i,'fire  ',j, 'fires ')
ENDIF
if (trace == '><') write(3,*) '<fire_s'
END
```

9.3 Fired_missile: Schedule Results of Firing a Missile

Fired_missile has a more difficult job. **Fire** calls it after a missile is fired and it finds whether another missile should be launched. **Impact** calls it when a missile impacts and in this case, it finds which of a number of events should be simulated.

Just after a missile is fired, the decision to fire another missile depends on these factors. If all guidance channels are in use, no more missiles may be fired until one of them impacts, so **fired_missile** is called at that time. If some guidance channels are idle, another missile may be fired almost immediately after the previous one is launched. In this case, **fired_missile** is called at firing time. In either case, **fired_missile** finds the time that the next missile is launched and schedules it.

The assumptions are:

1. If all guidance channels are busy, wait until one is free. This will occur when a missile impacts.
2. Do not move unless all guidance channels are free.
3. Do not switch weapons if a guidance channel is in use.

Any system that has no ammunition for any of its weapons will probably take cover. Helicopters and fixed wing will return to base though. For any system that has no effective ammunition, defenders will probably take cover and attackers may continue with their buddies with the expectation that they will find targets for their ammunition.

t_s	Simulation time (sec).
i, j	Firer, target.
do_pop	True if and only if pop-down criteria met.
do_switch	True if and only if target-switching criteria met.
no_msls	Ture if and only if missile supply exhausted.

```
SUBROUTINE FIRED_MISSILE (ts, i, j)
! Fired_missile: Move, fire, or switch targets after firing guided missile.
implicit none
real ts
integer i, j
logical no_msls, do_pop, do_switch
include 'global.h'

if (trace == '><') write(3,*) '>fire_m'
no_msls = nrd(i,1) == 0
do_pop = mision(i) == 'defend' .and. nrf(i) >= itbl(17,i)
do_switch = nrf(i) >= itbl(18,i)
IF (no_msls .or. do_pop .or. do_switch) THEN
   call diseng (ts,i,i,j,ISFOE)
ELSE
   call select (ts, i)
ENDIF
if (trace == '><') write(3,*) '<fire_m'
END
```

9.4 Fired_Burst: Schedule Results of Firing a Burst Round

Fired_burst schedules events that occur after a burst is fired or schedules the next round in the burst if the burst is incomplete. It is currently a dummy routine.

```
SUBROUTINE FIRED_BURST (ts,i,j,k)
!Fired_burst: Move, fire, or switch targets after firing a round in a burst. (NOT)
implicit none
real ts
integer i, j, k
include 'global.h'

if (trace == '><') write(3,*) '>fired_b'
call error ('Fired_burst: Not implemented. Firer is',i)
if (trace == '><') write(3,*) '<fired_b'
END
```

10. TERMINAL EFFECTS

The **impact** routine and the **damage** routine simulate terminal effects and their consequences.

10.1 Impact: Simulate Arrival of Round at the Target.

If the round is a guided missile, a guidance channel is freed and **fired_missile** is called to find what the firer does next. In any case, damage is assessed. **Damage** is called to find whether the target is killed.

t_s	Simulation time (s).
i	Firer
j	Target.
k_i	Kind of firer.
n_c	# of guidance channels.
k	Kind of round (or weapon).

```
SUBROUTINE IMPACT (ts,i,j)
!Impact: Simulate events that occur at impact.
implicit none
real ts
integer i, j
include 'global.h'
integer k, ki, nc

if (trace == '><') write(3,*) '>impact'
if (trace == 'impact') write(3,*) &
   'Impact: ts, i, j, e(j,1), kind(i)=', ts, i, j, e(j,1), kind(i)
i = abs(i)
if (e(j,1) == FD)  call error ('Impact: tgt in full defilade. Tgt j=', 0)
ki = kind(i)
nc = itbl(7,ki)
k = nwpn(i)
IF (k == 1 .and. nc > 0) THEN! Firer is using guided missiles.
   if (nrf(i) >= 2) call diseng (ts,i,i,j,ISFOE)
   call fired_missile (ts, i, j, ki)
ENDIF
call damage (ts,i,j,k)
if (trace == '><') write(3,*) '<impact'
END
```

10.2 Damage: Find if i Kills j

Damage finds whether firer i kills target j. It first finds the kind of round (or weapon) used and consults the appropriate table, interpolating based on the range from the firer to the target. The result is the probability of kill for that round, target, and range. It then draws a random number to find if the target was killed.

If j dies, **damage** cancels all events j was scheduled to perform, disengages all systems that were engaging j and counts the survivors on j's side. If there are no survivors on j's side, **damage** schedules the end of the battle (replication).

Arguments:

t_s	Simulation time (s).
i	Firer.
j	Target.

Locals:

k	Kind of round (or weapon).
p_{ks}	Probability of kill.

```
SUBROUTINE DAMAGE (ts,i,j,k)
!Damage: Find if target is killed and consequences.
implicit none
include 'global.h'
integer nsurv(2), ntgt(2)
logical over
real E_(7), ts, y1, y2, pks, ranu
integer i, j, k, ki, kj, L, LL, m, n
DATA E_ /0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0/
1 format(a16,2i5,a16,2i5)

if (trace == '><') write(3,*)  '>damage'
ki = kind(i)
kj = kind(j)
call  hunt(E_,7,r(i,j),LL)
y1 = p(LL,k,kj,ki)
y2 = p(LL+1,k,kj,ki)
pks = y1 + (r(i,j)-E_(LL)) * (y2-y1) / (E_(LL+1)-E_(LL))
PKS = 0.9
IF (ranu() < pks) THEN! Treat kill of target.
  alive(j) = .false.
   call history (i, j, ts, 'kills ', 'damage')
  if (kappa(i,j,13) == 0) kappa(i,j,13) = kappa(i,j,13) + 1
  kappa(i,j,10) = kappa(i,j,10) + 1
  call cancel (j,'all   ', 0, 'damage')
  call diseng(ts,1,is,j,ISDEAD)
! Count survivors on each side and each side's targets.
    nsurv(1) = 0
    nsurv(2) = 0
    ntgt(1) = 0
    ntgt(2) = 0
    DO L=1,is
      n = narmy(L)
      if (alive(L)) nsurv(n) = nsurv(n) + 1
      DO m=1,is
        if (lambda(L,m) == ISTGT) ntgt(n)=ntgt(n)+1
      END DO
    END DO
    if (ns <= 1) write(3,1)  '# survivors = ', nsurv, '# targets =',  ntgt
    over = nsurv(narmy(j))  ==  0 .and. ntgt(narmy(j)) == 0
    if (over) call cancel (0, 'all   ', 0, 'damage')
    if (over) call skedul (ts,0,'endsim',0, 'damage')
ELSE
  call history (j, 0, ts, 'survive', 'damage')
ENDIF
if (trace == '><') write(3,*)  '<damage'
END
```

11. TARGET DISENGAGEMENT

11.1 Diseng: Disengage Firer From Target

The relationship or coupling between each entity and every other entity changes from time to time during the course of a battle. At the beginning of each battle, all combatant is assumed to be outside the detection range of every other system. This is coupling ISOUT=1. Then each is checked to see if it is inside other combatants detection range. If so, the coupling between the searcher i and the target j is changed to INSIDE (2). This continues, with coupling ISTGT (7) being the highest coupling and ISDEAD (0) being the lowest coupling.

Diseng reduces the coupling between firer i and target j if necessary. Firer i may be a single system or it may be all systems except j. This is the most difficult routine in the entire model, so we will discuss it in fine detail.

Coupling between firer i and target j. The couplings are stored in the array λ. The table below shows the values and what they mean:

λ_{ij}	VALUE	COMMENT
0	ISDEAD	When j dies we set $\lambda_{ij} = \lambda_{ji} = 0$
1	ISOUT	Initial value. j is alive but assumed to be out of range.
2	INSIDE	At time zero, nearby friends are found to be in range. Foes generally come within detection range later.
3	INLOS	If j is within i's detection range, it may soon be in line-of-sight.
4	ISSEEN	This is set when i detects j and continues until ...
5	ISFRE	After detection, i classifies j as a friend.
6	ISFOE	After detection, i classifies j as a foe.
7	ISTGT	This is set when i decides to engage j.

Reasons for reducing the coupling.

The coupling is reduced if:
1. System j moves out of i's detection range.
2. Defender i pops down after firing a fixed number of rounds at j.
3. System i runs out of ammo.
4. System i switches targets after firing a fixed number of rounds at j
5. Defender j pops down.
6. System j moves out of line-of-sight.
7. System i and j diverge beyond i's detection range.
8. System j is killed.

Table 1 shows the routines that call the disengage routine, the new values in the λ array, and the reason for the change.

Table 1. The Effect of Events on Coupling

EVENT	λ_{ij}	λ_{ji}	REASON
Meet	ISOUT	unchanged	System j moved out of i's detection range.
Fire	ISFOE	unchanged	System i fired 2 rounds and has popped down.
Various*	ISFOE	unchanged	Multiple reasons
Damage	ISDEAD	ISDEAD	System j has been killed.
Pop down	$\leq$ INSIDE	$<$0$=$ INSIDE	System j has vanished.

* **Fired_single, fired_burst, fired_missile**, and **impact** all may call **diseng**. They call it because system i has fired a fixed number of rounds and is ready to pop down, because it has fired another fixed number of rounds and is ready to switch targets, or because it has run out of ammo and is ready to hide.

The logic is as follows:

SUBROUTINE DISENG (t_s, i_1, i_2, j, λ_n)
pick = .false.
IF (i is alive and $i \neq j$'s target) THEN
    Update connection between firer i and target j. (Must do before select!)
        $\lambda_{old} = \lambda_{ij}$
        if ($\lambda_{old} > \lambda_{new}$) $\lambda_{ij} = \lambda_{new}$
        if ($\lambda_{new} = INSIDE$ and $\lambda_{ji} > INSIDE$) $\lambda_{ji} = INSIDE$
        if ($\lambda_{new} = ISDEAD$) $\lambda_{ji} = ISDEAD$
    IF ($\lambda_{old} = ISTGT$) THEN
        Print that i drops j.
        Cancel all events for i against j
        IF (i is using guided missiles) THEN
            Release guidance channel
            If not guiding other missiles, schedule i to pop down for i.
            If guiding other missiles, attempt to select a tgt.
        ELSE IF (i has fired more than 1 round) THEN
            Schedule pop down for i
        ENDIF
        if (not guiding msl or not using msls) deselect weapon
    ENDIF
    if (Guidance channels idle or i is firing fire-and-forget rounds) deselect weapon.
    pick = stay_up
END IF

Arguments:

t_s	Simulation time (sec).
i_1	ID of first firer.

i_2	ID of last firer.
j	ID of system to be disengaged.
L_1	Tightest new coupling possible.

Key globals:

λ_{ij}	Coupling from firer i to target j
$alive_i$	True if and only if i is still functional.
$itbl_{7,ki}$	Number of guidance channels.
$nwpn_i$	ID of weapon i is using.
$nchan_i$	Number of busy guidance channels.
nrf_i	Number of rounds i has fired at current target.
μ_{i_μ,j_μ}	Count of changes from coupling i_μ to j_μ.

Locals:

i_μ	New coupling between i and j.
j_μ	New coupling between i and j.

```
SUBROUTINE DISENG (ts,i1,i2,j,Ln)
!Switch firers i1..i2 from j. Target j went out of range, vanished, or died.
include 'global.h'
logical guided
common /debug/ mu(8,8)

if (trace == '><') write(3,*) '>diseng'
if (Ln == ISDEAD) lambda(j,j) = ISDEAD
DO i=i1,i2
   imu = lambda(i,j) + 1
   IF (alive(i) .and. i /= j) THEN
      Lo = lambda(i,j)
      if (Lo > Ln) lambda(i,j) = Ln
      if (Ln == INSIDE .and. lambda(j,i) > INSIDE) lambda(j,i) = INSIDE
      if (Ln == ISDEAD) lambda(j,i) = ISDEAD
      if (Ln == ISDEAD) lambda(j,i) = ISDEAD
      IF (Lo == ISTGT) THEN! Disengage this target.
    call history (i, j, ts, 'drops  ', 'diseng ')
         call cancel(i,'all   ',j, 'diseng')
    guided = itbl(7,kind(i)) > 0 .and. nwpn(i) == 1
         IF (guided) THEN ! Treat missile systems.
            nchan(i) = nchan(i) - 1! Release guidance channel.
            IF (nchan(i) > 0) THEN! Still guiding another msl.
               call skedul (ts,i,'select',0,'diseng')
            ELSE
              call skedul (ts,i, 'popdn ',0, 'diseng')
            ENDIF
         ELSE IF (nrf(i) > 1) THEN! Pop-dn after firing 2 rounds?
            call skedul (ts,i,'popdn ',0,'diseng')
         END IF
         if (nchan(i) == 0 .or. nwpn(i) > 1) nwpn(i)=0
      END IF
   ENDIF
   jmu = lambda(i,j) + 1
   IF (imu /= jmu) mu(imu,jmu) = mu(imu,jmu) + 1
END DO
if (trace == '><') write(3,*) '<diseng'
```

```
END
!Ln    COMMENT
! 0 = ISDEAD j just died.
! 1 = ISOUT  j no longer in i's detection range. (One of them receded.)
! 2 = INSIDE j popped down, hid, or vanished (in det rg but not in LOS.)
! 6 = ISFOE  i fired 2 rds at j or ran out of (1? kind of) ammo.
! other      Error
```

12. TIME ADVANCE ROUTINES

The event routines are: **reset, skedul, event, and cancel. Reset** can be thought of as resetting the clock, clearing the calendar, or initializing the list of pending events. **skedul** inserts an event in chronological order while saving the time, ID of the entity performing the action, type of action, and possibly the ID of the entity receiving the action. **Event** fetches the next pending event, recovering the time, subject entity, action type, and object entity. **Cancel** removes zero or more events from the list.

12.1 Event Handling Using Linked Lists.

The two major ways of handling events are stepping a fixed time interval and stepping to the next significant event. Stepping to the next significant event (the method discussed here) requires routines to reset (initialize) the data structure, schedule an event, fetch an event, and cancel events. This section touches on various techniques for handling the event data and then discusses the linked list technique used by the software in the next four sections.

As a minimum, the model must store the time at which an event will occur, the identity of the entity that will perform the event, and the type of event. It may also be desirable to store the entity receiving the action of the event and other information about the event. If, at the current time t_s, routine **select** finds that after a delay of 5 seconds, Tank 4 may fire at Tank 6, this would require a Fortran call as follows:

call skedul (t_s+5.0,4,'fire ',6, 'select')

Methods of handling event data. A great many methods have been used for storing and retrieving event data. The simplest is to add an event to the end of a list and when the next event is needed, simply search the list for the event with the smallest time. The next simplest is to insert the event just before the next following event. This requires moving the next and all subsequent events down in the list and is slow. The method used here uses linked lists, so that the events are always sorted chronologically, but records of subsequent events need not be moved. (McCormack[2] discusses eight methods for handling event data applied to 12 problems. None of the 8 was fastest for all 12 problems, however, the method described here was best for 6 of them.)

The search from the front linear linked-list technique was used in the algorithms in the following sections. The key elements are:

- A set of links
- A pointer to the first idle link
- A pointer to the link containing the next event
- Several auxiliary pointers for manipulating the links

Initially, the *idle* pointer points to the first available link, which points to the subsequent link, and so on, until the last available link, which points to the null link Ω. The 'next event' pointer points to Ω also. When an event is inserted in the list, the algorithm removes the first idle link from the chain of idle links, inserts it chronologically in

the chain of active links, and inserts the event data into the link.

Retrieving the next event simply involves copying the data from the first link of the chain of active links, removing the link from that chain, and inserting it at the head of the chain of idle links.

Cancelling an event is similar, but involves links anywhere in the chain of active links. This implementation stores as many as 200 events. Each type of information is stored in an array dimensioned to 200, however a given link consists of the ith element of each array. The arrays are:

real when(200)	Time of the event
integer who(200)	The entity performing the event
character*6 what(200)	The type of event performed
integer whom(200)	The entity receiving the event
integer next(200)	The pointer to the next link

when(9)	who(9)	what(9)	whom(9)	next(9)
18.32	4	fire..	6	31

Figure 2. Contents of a Link

12.2 Reset: Re-initialize the Event List.

Reset 'resets the clock' to time zero. To do this, it rebuilds the linked list of idle events and clears the linked list of active events. It is one of four routines, **reset, skedul, cancel, and event**, that cooperate to handle events in Monte Carlo Simulations. Although it was designed for use in combat simulations, it has much broader use.

If the single argument to **reset** is true, the event routines will print out each event as it is scheduled or cancelled; if false, this printing is not done. The subroutine then builds a linked list of idle links, as shown at the top of Figure 3. It also makes a null linked list of active links as shown at the bottom of Figure 3 no events are yet scheduled.

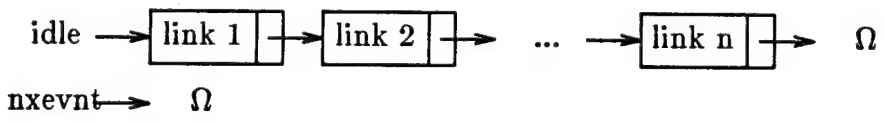


Figure 3. The Initial Linked Lists

Globals:

who	Active combatant.
whom	Target.
prflag	True if and only if clock routines should print their activities.
what	Event that is scheduled.

when	Time of event (often seconds).
$next_i$	Link to next most immanent event.
nxevnt	Pointer to most immanent event.
nxidle	Pointer to first idle node.

```
      ! clock.h file
parameter (NE=200)
character(len=6) what
integer who, whom
logical prflag
common /event1/ what(NE)
common /event2/ when(NE), who(NE), whom(NE), next(NE), nxevnt, nxidle, prflag
save /event1/, /event2/
```

Argument:

prflg	True if and only if clock routines are to print their activities.

Local:

j	Node/event index.

```
SUBROUTINE RESET (prflg)
!Reset: Initialize the clock to time zero.
include 'clock.h'
logical prflg

prflag = prflg
nxevnt = 0
nxidle = 1
DO j=1,NE
  next(j) = j+1
END DO
next(NE) = 0
END
```

12.3 Skedul: Schedule an Event.

The **skedul** subroutine schedules an event in a linked list of events. The event information stored is; event type, entity that will perform the event, time the event will occur, and perhaps the receiver of the action.

The calling statement. The arguments to the **skedul** subroutine tell when, who, what, and whom. That is when will a future (tentative) event occur, which entity will perform that event, what event (activity) will be performed and possibly to whom will that activity be directed. If, for example, the combatant i will fire 12 seconds in the future at system j then the following statement would appear in the program:

call skedul (t_s+t_f, i,'fire ', j)

Where:

t_s is the current time,

t_f is the time delay after which the event may occur,
i is the subject or actor causing the event,
'fire ' is a six-character string identifying the type of event, and
j is an integer identifying the object of the event.

Note that the event must always occur in the future, so in the example, $t_f > 0.0$.

When **skedul** is called like this, it inserts the when, who, what, and whom data into a linked list in chronological order with other scheduled events. In this case, the time to fire is the current time t_s plus 12 seconds, the actor is combatant i, the event is indicated by the six-character string stored in the 'fire ' variable, and the target (the whom) is indicated by an integer stored in the j variable.

Algorithm. On average, **skedul** must traverse half the linked list to find the place to insert the event link. It must also check to see if an idle link is available. If so, it then inserts the new event using these 6 steps, as shown in Figure 4.

1. Store the index of the idle link/event in n.
2. Store the index of the new head of the idle chain in *idle*.
3. Store the index of the immediately preceding link/event in l.
4. Store the index of the succeeding idle link/event in m.
5. Store the index of the now active link/event in $next_l$.
6. Store the index of the succeeding link/event in the now active link/event in $next_n$.

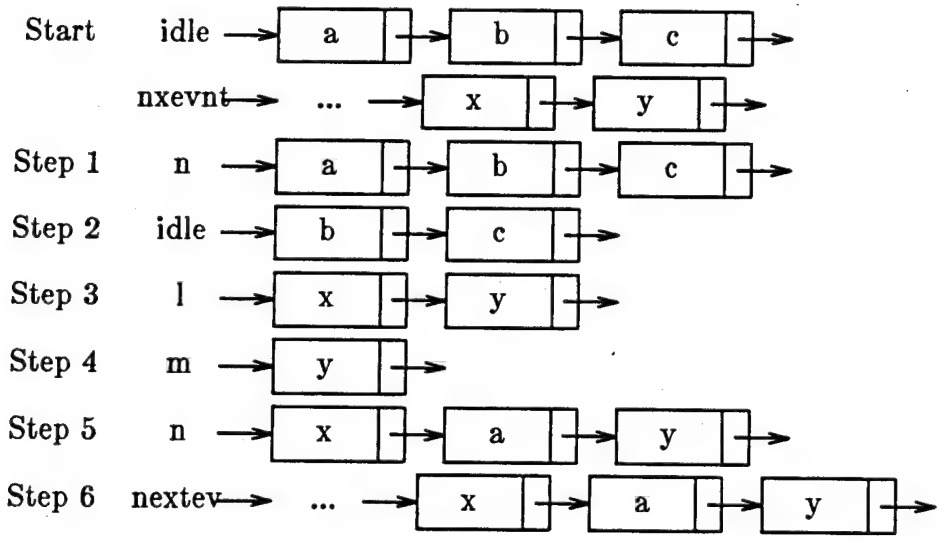


Figure 4. Scheduling an **event**

Arguments:

t	Time event occurs.
i	Actor, combatant
act	Event type.
j	Receiver of action, target.
caller	Name of calling routine.

Locals:

jj	Node, event index. Used only when too many events for the nodes allocated.
n	Temporary pointer to next idle node.
l	Temporary pointer to preceding node that is in use.
m	Temporary pointer to following node that is in use.

```
      SUBROUTINE SKEDUL (t,i,act,j, caller)
! 9Schedule: Schedule an event for later execution.
include 'clock.h'
character act*6, caller*6
1 format(2i3, 8x, f8.2, ' +', a6, ' ',a6)
2 format (i5,g20.3, 2i2, 1x, a)

if (prflag )write(3,1) i, j, t, act, caller
IF (abs(t) > 3000.0) write(3,1) i, j, t, act, caller
IF (abs(t) > 3000.0) STOP
IF (nxidle == 0) THEN ! stop cause storage all used up.
  write(3,*)' Storage overloaded with too many events.'
  jj = nxevnt
  DO
    write(3,2) next(jj), when(jj), who(jj), whom(jj), what(jj)
    jj = next(jj)
  IF (jj == 0) EXIT
  END DO
  jj = nxidle
  DO
    write(3,2) next(jj), when(jj), who(jj), whom(jj), what(jj)
    jj = next(jj)
  IF (jj == 0) EXIT
  END DO
  STOP
ELSE ! Store the event
  ! Cut storage unit from empties
    n = nxidle
    nxidle = next(nxidle)
  ! Then find where to insert this event in the event list.
  IF (nxevnt <= 0) THEN ! New event is only event
    next(n) = 0
    nxevnt = n
  ELSE ! Then find where to insert it.
  !  Point to first 2 events
       l = nxevnt
       m = next(l)
  !  Find where to insert them
      IF (t >= when(l)) THEN ! See if between 2 scheduled events.
      ! Loop till found.
          DO
      IF (m == 0) EXIT
      IF (t < when(m)) EXIT
            l = m
            m = next(m)
      END DO
       ! Splice new event into list
         next(n) = m
         next(l) = n
    ELSE ! Place new event as most imminent
      next(n) = nxevnt
      nxevnt = n
    ENDIF
  ENDIF
```

```
! Store event info
    when(n) = t
    what(n) = act
    who(n) = i
    whom(n) = j
ENDIF
END
```

12.4 Event: Find Next Event.

The **event** subroutine finds the next event to be simulated from a linked list of events.

The **event** subroutine is only called when an event is completed and the simulation is ready to execute the next event at the top of the list. One of the 'model' routines called **events** is the only routine that calls **event**. It is called as follows:

$$\text{call event}(t_s, i, \text{act}, j,)$$

All four arguments are output from **event** and contain the time of the most imminent event, who (which tank) is performing the event, what event is being performed, and whom (which target) is receiving the action. If t_s, i, *act* have the values 10.5, 4, 'select' then the current becomes 10.5 seconds and at that time Tank 4 attempts to select a target. (The variable j is undefined for this particular event.)

The event routine simply extracts the information for the next event from the first link on the linked list of events and then moves that link to the head of the linked list of idle links. The information extracted is:

i - the entity performing the event
act - the event or act
j - the object of the event (or other useful information)
t_s - the time the event occurs

Figure 5 shows the arrangement of the idle and active linked lists before and after the most imminent event is fetched.

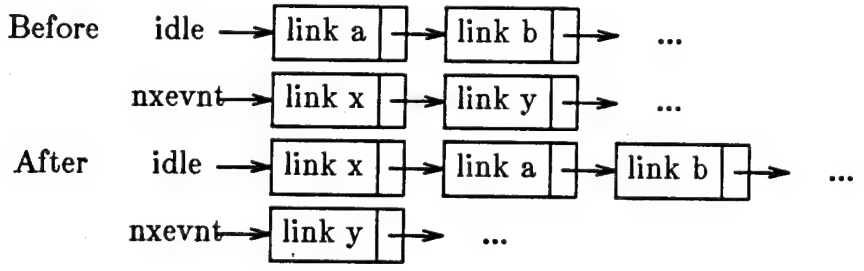


Figure 5. Selecting the next event.

Arguments:

t	Time event occurs.

i	Actor, combatant
act	Event type.
j	Receiver of action, target.

```
      SUBROUTINE EVENT (t,i,act,j)
! Event: Find the next scheduled event.
include 'clock.h'
character*6 act

! Fill arguments
    i = who(nxevnt)
    act = what(nxevnt)
    j = whom(nxevnt)
    t = when(nxevnt)
! Drop storage unit from active storage chain
    n = nxevnt
    nxevnt = next(nxevnt)
! Add storage unit to inactive storage.
    next(n) = nxidle
    nxidle = n
END
```

12.5 Cancel: Cancel an Event.

The **cancel** subroutine cancels an event from a linked list of events.

Cancel removes zero or more links (events) from the list of scheduled events and places them in the linked list of idle links. This removes the record of these events, so they never occur. **Cancel** is called in the four ways illustrated below:

```
call cancel (i,'fire  ', j)
call cancel (i,'all   ', j)
call cancel (i,'all   ',NULL)
call cancel (i,'fire  ',NULL)
```

The first call to **cancel** cancels any fire events associated with entity *i* and object *j*. The second version cancels all events associated with entity *i* and object *j*. The third version cancels all events associated with entity *i*, no matter what is the object of the action. The fourth version cancels all fire events associated with entity *i*.

Figure 6 shows how the active and idle chains look before and after cancelling the second active event, event y.

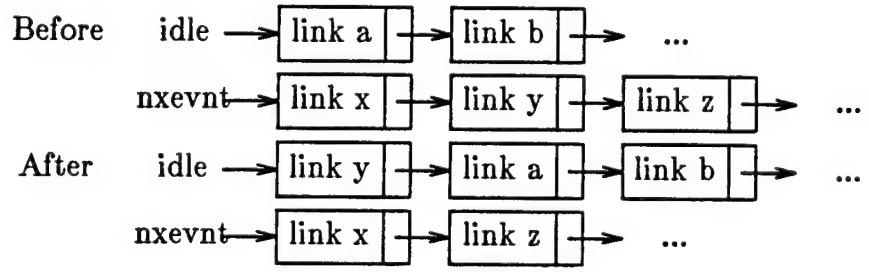


Figure 6. Cancelling an event.

Arguments:

t	Time event occurs.
i	Actor, combatant
act	Event type.
j	Receiver of action, target.
caller	Name of calling routine.

Locals:

iswho	True if and only if entity of event to be cancelled.
iswhat	True if and only if event to be cancelled.
iswhom	True if and only if target for event to be cancelled.

```
      SUBROUTINE CANCEL (i, act, j, caller)
!Cancel: cancel 'act' events for 'i' entity.
!  (all events if act='')
!Definitions of local variables:
!  m - pointer to previous event
!  n - pointer to current event being considered
include 'clock.h'
logical iswhat, iswho, iswhom
character*6 act, caller
1 format(2i3, 8x, f8.2, ' -', a6, ' ',a6)

m = 0
n = nxevnt
DO WHILE (n>.0)
  iswho  = i ==who(n)
  iswhat = act==what(n) .or. act=='all   '
  iswhom = j==whom(n) .or. j==0
  IF (iswho .and. iswhat .and. iswhom) THEN ! Then remove event
    if (prflag )write(3,1) i, j, when(n), what(n), caller
    if (m==0) nxevnt = next(n)
    if (m /= 0) next(m) = next(n)
    next(n) = nxidle
    nxidle = n
    if (m==0) n = nxevnt
    if (m /= 0) n = next(m)
  ELSE ! Move to next event.
    m = n
    n = next(n)
  ENDIF
ENDDO
END
```

13. UTILITY ROUTINES

13.1 History: Print Event History if Only One Replication

Below is a a portion of an event history showing the output of **history**. Column 1 gives the active entity (firer, searcher, etc) Column 2 gives the target or 0 if no specific target is used. Column 3 gives the simulation time (sec). Column 4 tells what happens at that time. Column 5 tells what routine simulated the event.

For example the first line says that **bound** rescheduled itself for Entity 3. Then **events** called a bound event for Entity 2 at time 960 sec. Then **bound** actually simulated entity 2 entering overwatch.

```
     :
 3   0                  bound   bound
 2   0   960.00         bound   events
 2   0                  owatch  bound
 1   2                  diverg  meet
 1   3                  convrg  meet
 1   3                  have    los
 2   3                  convrg  meet
 2   3                  have    los
 3   0   960.00         vanish  events
 3   0                  doesnt  vanish
 1   3                  detect  search
 1   3                  enemy!  clasif
 1   3                  select  select
 1   3   962.67         popup   events
 1   3                  does    popup
 3   2                  detect  search
 1   3   973.22         fire    events
 1   3                  fire @  fire
 1   3   973.97         impact  events
 1   3                  (have)  impact
 1   3                  kills   damage
 1   3                  drops   diseng
 1   0   973.97         popdn   events
 1   0                  does    popdn
     :
```

Arguments:

i, *j* Firer, target

t_s Simulation time (s).

what Event type.

caller Name of calling routine.

```
SUBROUTINE HISTORY (i, j, ts, what, caller)
! History: Print event history if only one replication.
implicit none
include 'global.h'
character (len=*) what, caller
integer i, j; real ts
1 format (2i3,f8.2,10x,a6,' ',a6)
2 format (2i3,18x,a6,' ',a6)
```

```
IF (nm == nn) THEN ! (This is the battle to monitor.)
   if (caller == 'events') write(3,1) i, j, ts, what, caller
   if (caller /= 'events') write(3,2) i, j,     what, caller
ENDIF
END
```

13.2 Error: Print Error Message and Stop

Error prints its arguments, a character string, and an integer. Then it stops. The string tells why the program stopped and the integer gives a clue, too. Here are the error messages:

Impact: tgt in full defilade 0
'MOVE: no data for:'//namei(i) i
SEARCH: L must be 1..7. L= L

```
SUBROUTINE ERROR (str,i)! Print line and stop.
character (len=*) str

write(3,*) 'Error: i=', i
write(6,*) str, i
STOP
END
```

13.3 Ranu: Draw a Pseudo-random Number Between 0, 1.

Ranu is a version of the uran31 random uniform number generator[1]. It prints values between values including 0. but not including 1.0.

```
FUNCTION RANU ()
!Ranu: A version of uran31 random uniform nr generator.
common /crandm/ i
real a1
j=i
j=j*25
j=j-(j/67108864)*67108864
j=j*25
j=j-(j/67108864)*67108864
j=j*5
j=j-(j/67108864)*67108864
a1=j
i=j
ranu= a1/67108864
END
```

1. Pike, M. C., and Hill, I. D., "Algorithm 266 Pseudo-Random Numbers," page 266-P 1- 0, "Remark on Algorithm 266, Pseudo-Random Numbers," page 266-P 2- R1, "*Collected Algorithms from CACM,*" volume 11, Association for Computing machinery, NY, 1980.

13.4 Rolln: Draw Two Gaussian Deviates

```
      FUNCTION ROLLN(sigma)
! Draw a random number from a normal dist w/ Box-Muller method.
save j, z
data j/0/

IF (j == 0) THEN
  x = sqrt(-2.*alog(ranu()))
  y = 2.*3.1415926535*ranu()
  rolln = x*cos(y)*sigma
  z = x*sin(y)
ELSE
  j = 1-j
  rolln = z*sigma
ENDIF
END
```

13.5 Hunt: Find the Index for Interpolation

Hunt uses its arguments, the vector **x** and the scalar x_d to find the index i such that $x_i \leq x_d < x_{i+1}$. The code may be found in *Numerical Recipes*, by Press, et al. It uses a variant of binary search that is optimized for repeated use when the output is expected to be the same or nearly the same as on the previous call to the routine.

13.6 Ialf: Find the Location of a Character String in a List

Ialf takes a character string argument and finds its location in a list of such character strings. If the M1A1 tank is the second system described in input data, then **ialf** will find 'M1A1' in the second position on the list. So if the eighth system is an M1A1, a call to **ialf** will show it is the second kind on the list and System 8 will be of Kind 2. The program then looks up data for System 8 in the data for system Kind 2.

Arguments:

words	A list of as many as 10 words, each as long as six characters long.
n	The actual number of entries in the list.
word	The string to locate in the list.
ialf	The location of the word

Local:

i	The location being checked.

```
FUNCTION IALF (words,n,word)
!Ialf - Find location of a word in a list of words.
parameter (NX=10)
character*6 words(NX), word

i = n + 1
```

```
DO
   i = i - 1
IF (i == 0) EXIT
IF (words(i) == word) EXIT
END DO
ialf = i
END
```

NO. OF COPIES	ORGANIZATION
2	ADMINISTRATOR DEFENSE TECHNICAL INFO CTR ATTN DTIC DDA CAMERON STATION ALEXANDRIA VA 22304-6145
1	DIRECTOR US ARMY RESEARCH LAB ATTN AMSRL OP SD TA 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR US ARMY RESEARCH LAB ATTN AMSRL OP SD TL 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR US ARMY RESEARCH LAB ATTN AMSRL OP SD TP 2800 POWDER MILL RD ADELPHI MD 20783-1145
	ABERDEEN PROVING GROUND
5	DIR USARL ATTN AMSRL OP AP L (305)

NO. OF COPIES	ORGANIZATION
1	HQDA ATTN SARD TT DR F MILTON PENTAGON WASHINGTON DC 20310-0103
1	PROJECT MANAGER TANK MAIN ARMAMENT SYSTEMS ATTN SFAE AR TMA PICATINNY ARSENAL NJ 07806-5000
1	PROJECT MANAGER TANK MAIN ARMAMENT SYSTEMS ATTN SFAE AR TMA MS PICATINNY ARSENAL NJ 07806-5000
1	PROJECT MANAGER TANK MAIN ARMAMENT SYSTEMS ATTN SFAE AR TMA PA PICATINNY ARSENAL NJ 07806-5000
1	COMMANDER US ARMY ARDEC ATTN SMCAR FSF BD PICATINNY ARSENAL NJ 07806-5000
1	COMMANDER US ARMY ARDEC ATTN SMCAR AEF CD PICATINNY ARSENAL NJ 07806-5000
1	COMMANDER US ARMY ARDEC ATTN SMCAR CCH T PICATINNY ARSENAL NJ 07806-5000
1	PROJECT MANAGER ABRAMS TANK SYSTEM ATTN SFAE ASM ABS S WARREN MI 48397-5000
4	COMMANDER US ARMY ARMOR CTR & FT KNOX ATTN ATSB CDM ATSB TSM ATZK TS ATZK CD FT KNOX KY 40121-5212

NO. OF COPIES	ORGANIZATION
1	COMMANDER US ARMY TANK AUTOMOTIVE CMD ATTN AMSTA TR R AVT MS MILANOV WARREN MI 48397-5000
1	PROJECT MANAGER ARMORED GUN SYSTEM ATTN SFAE AG WARREN MI 48397-5000
	ABERDEEN PROVING GROUND
7	DIR USARL ATTN: AMSRL-WT-W, C. MURPHY AMSRL-MO-IF, S. CONNEALLY AMSRL-WT-PB, E. SCHMIDT AMSRL-WT-WA, B. MOORE AMSRL-WT-WE, J. LACETERA AMSRL-WT-WE, G. SAUERBORN J. WALD
5	DIR, USAMSAA ATTN: AMXSY-EA, R. NORMAN AMXSY-GA, W. BROOKS AMXSY-GC, G. COMSTOCK M. SCHMIDT L. HARRINGTON
1	CDR, USACSTA ATTN: STECS-AE-TF, M. HENRY

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number ARL-TR-883 Date of Report October 1995

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)